

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/lm_diags.c

DATE 5/23/89

PAGE #

TIME 4:41:14 pm

9/20

```

LINE # SOURCE TEXT
961 /* returns the interrupt character if in the buffer. */
962 /* or returns the first character in the buffer */
963 /* or returns 0 if no key */
964
965
966 _check_key()
967 {
968     register int retval;
969
970 #ifdef CPU_DIAGS
971     if (fifo_task == RECEIVE_TASK_ID) {
972         retval = _net_check_key();
973     } else
974     {
975         retval = _serial_check_key();
976     }
977 #else CPU_DIAGS
978     retval = _serial_check_key();
979 #endif CPU_DIAGS
980     if (retval == INTR_CHARACTER) intr();
981     return retval;
982 }
983
984 _serial_check_key()
985 {
986 #ifdef CPU_DIAGS
987     struct fifo_entry fifo;
988     register int retval;
989     register int i;
990
991     fifo.fifo_no = RX_FIFO;
992     retval = fifo_inquiry(&fifo);
993
994     if (retval) {
995         retval = _serial_get_key();
996         while (fifo_inquiry(&fifo)) {
997             i = _serial_get_key();
998             if (i == INTR_CHARACTER)
999                 retval = i;
1000         }
1001     }
1002     if (i == ctrl(s)) {
1003         (void) _serial_get_key();
1004         /* simple ^S strategy: any char resumes printing */
1005     }
1006     return retval;
1007 #else CPU_DIAGS
1008     register int retval = _check_key();
1009     register int i;
1010
1011     if (retval) {
1012         retval = _get_key();
1013         while (_check_key()) {
1014             i = _get_key();
1015             if (i == INTR_CHARACTER)
1016                 retval = i;
1017         }
1018         if (i == ctrl(s)) (void) _get_key(); /* simple ^S strategy:
1019                                             any char resumes printing */
1020     }
1021     return retval;
1022 #endif CPU_DIAGS
1023 }
1024
1025 /* parameter parsing routines */
1026
1027 #define CMDTOKEN_NOT_BUILT_IN 0
1028 #define CMDTOKEN_quit 1
1029 #define CMDTOKEN_all 2
1030 #define CMDTOKEN_exit 3
1031 #define CMDTOKEN_help 4
1032 #define CMDTOKEN_main 5
1033 #define CMDTOKEN_wizard 6
1034 #define CMDTOKEN_burnin 7
1035
1036 #define TOKEN_UNDEFINED 0
1037 #define TOKEN_max_cycles 1
1038 #define TOKEN_max_errors 2
1039 #define TOKEN_max_warnings 3
1040 #define TOKEN_max_messages 4
1041 #define TOKEN_summary_count 5
1042 #define TOKEN_errors_on 6
1043 #define TOKEN_warnings_on 7
1044 #define TOKEN_messages_on 8
1045 #define TOKEN_banner_on 9
1046 #define TOKEN_print_depth 10
1047 #define TOKEN_lm_fast_test 11
1048 #define TOKEN_forever 12
1049 #define TOKEN_output 13
1050 #define TOKEN_COUNT 14
1051 #define TOKEN_max_failures 15
1052
1053 #define DFLT_max_cycles 1
1054 #define DFLT_max_errors 1
1055 #define DFLT_max_failures 0
1056 #define DFLT_max_warnings 0
1057 #define DFLT_max_messages 0
1058 #define DFLT_summary_count 0
1059 #define DFLT_errors_on LM_FALSE
1060 #define DFLT_warnings_on LM_FALSE
1061 #define DFLT_messages_on LM_TRUE
1062 #define DFLT_banner_on LM_TRUE
1063 #define DFLT_print_depth 20
1064 #define DFLT_lm_fast_test LM_TRUE
1065 #define DFLT_forever LM_FALSE
1066
1067 char *get_identifier();
1068
1069 struct token {
1070     char *name;
1071     long length;
1072     long token;
1073 };
1074
1075 struct token cmdtokens[] = {
1076     { "all", 1, CMDTOKEN_all },
1077     { "burnin", 4, CMDTOKEN_burnin },
1078     { "quit", 1, CMDTOKEN_quit },
1079     { "exit", 1, CMDTOKEN_exit },
1080     { "help", 1, CMDTOKEN_help },

```

| | | | | | |
|--|--|------------------------------------|--|-----------------|-----------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/lm_diags.c | | DATE 5/23/89 | PAGE # 10/21 |
| TIME 4:41:14 pm | | | | | |

| LINE # | SOURCE TEXT | | | |
|--------|--|----|-----------------------|--|
| 1081 | { "main", | 1, | CMTOKEN_main}, | |
| 1082 | { "wizard", | 3, | CMTOKEN_wizard}, | |
| 1083 | 0, | | | |
| 1084 | 1, | | | |
| 1085 | } | | | |
| 1086 | struct token new_options[] = { | | | |
| 1087 | { "continuous", | 1, | TOKEN_forever}, | |
| 1088 | { "error_count", | 1, | TOKEN_max_errors}, | |
| 1089 | { "failure_count", | 1, | TOKEN_max_failures}, | |
| 1090 | { "fast_test", | 4, | TOKEN_lm_fast_test}, | |
| 1091 | { "message_count", | 1, | TOKEN_max_messages}, | |
| 1092 | { "print_banner", | 7, | TOKEN_banner_on}, | |
| 1093 | { "print_depth", | 7, | TOKEN_print_depth}, | |
| 1094 | { "print_errors", | 7, | TOKEN_errors_on}, | |
| 1095 | { "print_messages", | 7, | TOKEN_messages_on}, | |
| 1096 | { "print_warnings", | 7, | TOKEN_warnings_on}, | |
| 1097 | { "p_banner", | 2, | TOKEN_banner_on}, | |
| 1098 | { "p_depth", | 2, | TOKEN_print_depth}, | |
| 1099 | { "p_error", | 2, | TOKEN_errors_on}, | |
| 1100 | { "p_message", | 2, | TOKEN_messages_on}, | |
| 1101 | { "p_warning", | 2, | TOKEN_warnings_on}, | |
| 1102 | { "repeat", | 1, | TOKEN_max_cycles}, | |
| 1103 | { "summary_print", | 1, | TOKEN_summary_count}, | |
| 1104 | { "warning_count", | 1, | TOKEN_max_warnings}, | |
| 1105 | 0 | | | |
| 1106 | }; | | | |
| 1107 | | | | |
| 1108 | set_global_defaults() | | | |
| 1109 | { | | | |
| 1110 | max_cycles = DFLT_max_cycles; | | | |
| 1111 | max_errors = DFLT_max_errors; | | | |
| 1112 | max_failures = DFLT_max_failures; | | | |
| 1113 | max_warnings = DFLT_max_warnings; | | | |
| 1114 | max_messages = DFLT_max_messages; | | | |
| 1115 | summary_count = DFLT_summary_count; | | | |
| 1116 | errors_on = DFLT_errors_on; | | | |
| 1117 | warnings_on = DFLT_warnings_on; | | | |
| 1118 | messages_on = DFLT_messages_on; | | | |
| 1119 | banner_on = DFLT_banner_on; | | | |
| 1120 | print_depth = DFLT_print_depth; | | | |
| 1121 | lm_fast_test = DFLT_lm_fast_test; | | | |
| 1122 | forever = DFLT_forever; | | | |
| 1123 | } | | | |
| 1124 | | | | |
| 1125 | /* execute_string() returns FAILURE only if we want to quit */ | | | |
| 1126 | | | | |
| 1127 | execute_string(string, menu) | | | |
| 1128 | register char *string; | | | |
| 1129 | LM_DIAG_MENU *menu; | | | |
| 1130 | #define is_specified(token) (specified & (1 << (token))) | | | |
| 1131 | { | | | |
| 1132 | INTR_INIT | | | |
| 1133 | register LM_DIAG_MENU_ITEM *menu_item; | | | |
| 1134 | long value, cmdtoken, token; | | | |
| 1135 | long reply; | | | |
| 1136 | char command[max_str]; | | | |
| 1137 | long specified = 0; /* nothing specified yet */ | | | |
| 1138 | | | | |
| 1139 | set_global_defaults(); | | | |
| 1140 | | | | |
| 1141 | if (! (string = get_identifier(string, command))) | | | |
| 1142 | return SUCCESS; | | | |
| 1143 | | | | |
| 1144 | cmdtoken = identify_token(command, &(cmdtokens[0])); | | | |
| 1145 | | | | |
| 1146 | while (string = new_get_parameter(string, &token, &value)) { | | | |
| 1147 | { | | | |
| 1148 | case TOKEN_max_cycles: max_cycles = value; break; | | | |
| 1149 | case TOKEN_max_errors: max_errors = value; break; | | | |
| 1150 | case TOKEN_max_failures: max_failures = value; break; | | | |
| 1151 | case TOKEN_max_warnings: max_warnings = value; break; | | | |
| 1152 | case TOKEN_max_messages: max_messages = value; break; | | | |
| 1153 | case TOKEN_summary_count: summary_count = value; break; | | | |
| 1154 | case TOKEN_errors_on: errors_on = value; break; | | | |
| 1155 | case TOKEN_warnings_on: warnings_on = value; break; | | | |
| 1156 | case TOKEN_messages_on: messages_on = value; break; | | | |
| 1157 | case TOKEN_banner_on: banner_on = value; break; | | | |
| 1158 | case TOKEN_print_depth: print_depth = value; break; | | | |
| 1159 | case TOKEN_lm_fast_test: lm_fast_test = value; break; | | | |
| 1160 | case TOKEN_forever: forever = value; break; | | | |
| 1161 | case TOKEN_UNDEFINED: | | | |
| 1162 | lm_banner("Syntax error: %s", string); | | | |
| 1163 | return SUCCESS; | | | |
| 1164 | | | | |
| 1165 | default: | | | |
| 1166 | lm_banner("Token %d not implemented", token); | | | |
| 1167 | return SUCCESS; | | | |
| 1168 | } | | | |
| 1169 | specified = (1 << token); | | | |
| 1170 | } | | | |
| 1171 | | | | |
| 1172 | /* | | | |
| 1173 | Defaults: | | | |
| 1174 | 1. Turn on all messages, warnings, errors only if a utility or | | | |
| 1175 | you are a wizard running a single test with no reps. | | | |
| 1176 | 2. Turn off banners for multiply associated alltests. | | | |
| 1177 | 3. pb-f implies pb-i implies ...f implies pb-f | | | |
| 1178 | 4. pb-t implies pb-t implies pb-t implies pb-t | | | |
| 1179 | (overrides 3 above) | | | |
| 1180 | */ | | | |
| 1181 | | | | |
| 1182 | /* See 1. above */ | | | |
| 1183 | if (lm_wizard_mode && (max_cycles <= 1) && | | | |
| 1184 | (forever == LM_FALSE) && (cmdtoken != CMTOKEN_all)) { | | | |
| 1185 | if (!is_specified(TOKEN_errors_on)) errors_on = LM_TRUE; | | | |
| 1186 | if (!is_specified(TOKEN_warnings_on)) warnings_on = LM_TRUE; | | | |
| 1187 | if (!is_specified(TOKEN_messages_on)) messages_on = LM_TRUE; | | | |
| 1188 | } | | | |
| 1189 | | | | |
| 1190 | /* 2. Default pb = f if r > 1 */ | | | |
| 1191 | if ((max_cycles > 1) (forever == LM_TRUE)) { | | | |
| 1192 | if (!is_specified(TOKEN_banner_on)) banner_on = LM_FALSE; | | | |
| 1193 | } | | | |
| 1194 | | | | |
| 1195 | /* 3. pb-f implies pb-i implies ...f implies pb-f */ | | | |
| 1196 | if (!is_specified(TOKEN_banner_on) && (banner_on == LM_FALSE)) { | | | |
| 1197 | if (!is_specified(TOKEN_messages_on)) messages_on = LM_FALSE; | | | |
| 1198 | } | | | |
| 1199 | | | | |
| 1200 | /* | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/lm_diags.c

DATE 5/23/89
TIME 4:41:14 pm

PAGE #
11/22

```

1201         if (!is_specified(TOKEN_warnings_on)) warnings_on = LM_FALSE;
1202         if (!is_specified(TOKEN_errors_on)) errors_on = LM_FALSE;
1203     }
1204     if (is_specified(TOKEN_errors_on) && (errors_on == LM_FALSE)) {
1205         if (!is_specified(TOKEN_messages_on)) messages_on = LM_FALSE;
1206         if (!is_specified(TOKEN_warnings_on)) warnings_on = LM_FALSE;
1207     }
1208     if (is_specified(TOKEN_warnings_on) && (warnings_on == LM_FALSE)) {
1209         if (!is_specified(TOKEN_messages_on)) messages_on = LM_FALSE;
1210     }
1211
1212     /* 4. port implies port implies port implies port */
1213     if (is_specified(TOKEN_messages_on) && (messages_on == LM_TRUE)) {
1214         if (!is_specified(TOKEN_banner_on)) banner_on = LM_TRUE;
1215         if (!is_specified(TOKEN_errors_on)) errors_on = LM_TRUE;
1216         if (!is_specified(TOKEN_warnings_on)) warnings_on = LM_TRUE;
1217     }
1218     if (is_specified(TOKEN_warnings_on) && (warnings_on == LM_TRUE)) {
1219         if (!is_specified(TOKEN_banner_on)) banner_on = LM_TRUE;
1220         if (!is_specified(TOKEN_errors_on)) errors_on = LM_TRUE;
1221     }
1222     if (is_specified(TOKEN_errors_on) && (errors_on == LM_TRUE)) {
1223         if (!is_specified(TOKEN_banner_on)) banner_on = LM_TRUE;
1224     }
1225
1226     switch (cmdtoken) {
1227     case CMDTOKEN_quit:
1228         return FAILURE; /* quit */
1229     case CMDTOKEN_all:
1230         lm_all_test(menu);
1231         break;
1232     case CMDTOKEN_banner:
1233         lm_banner("Banner is:\n");
1234         if (!is_specified(TOKEN_forever)) forever = LM_TRUE;
1235         if (!is_specified(TOKEN_errors_on)) errors_on = LM_TRUE;
1236         if (!is_specified(TOKEN_messages_on)) messages_on = LM_FALSE;
1237         if (!is_specified(TOKEN_warnings_on)) warnings_on = LM_FALSE;
1238         if (!is_specified(TOKEN_banner_on)) banner_on = LM_FALSE;
1239         lm_all_test(menu);
1240         break;
1241     case CMDTOKEN_exit:
1242         exit_request();
1243         break;
1244     case CMDTOKEN_help:
1245         help();
1246         break;
1247     case CMDTOKEN_wizard:
1248         promote();
1249         break;
1250     case CMDTOKEN_main:
1251         lm_pop_up = LM_TRUE;
1252         return FAILURE; /* quit */
1253     default: /* not a built in command */
1254         menu_item = menu->menu_items;
1255         for (reply = 0;
1256             reply <= number_of_items; ++reply, ++menu_item) {
1257             if (0 == strcmp(cmdtoken, menu_item->selection))
1258                 break;
1259         }
1260
1261         if ((reply == menu->number_of_items)
1262             || ((menu_item->attributes & LM_DIAG_wizard_mode)
1263                 && !lm_wizard_mode)
1264             || ((menu_item->attributes & LM_DIAG_no_select)) {
1265             menu_line(LM_DIAG_ERROR, "Invalid selection: %s\n",
1266                     cmdtoken);
1267             break;
1268         }
1269         if ((menu_item->attributes & LM_DIAG_dflt_msg_off)) {
1270             /* a utility */
1271             if (!is_specified(TOKEN_errors_on)) errors_on = LM_TRUE;
1272             if (!is_specified(TOKEN_warnings_on)) warnings_on = LM_TRUE;
1273             if (!is_specified(TOKEN_messages_on)) messages_on = LM_TRUE;
1274         }
1275         menu->current_selection = reply;
1276         INTR_BEGIN {
1277             if (menu->menu_items[menu->current_selection].attributes
1278                 & LM_DIAG_no_repeat) {
1279                 executeRoutine(menu, 1); /* no cwt or xpl */
1280             } else {
1281                 if (LM_TRUE == forever) {
1282                     run_cmd(menu);
1283                     break;
1284                 } else {
1285                     executeRoutine(menu, max_cycles);
1286                 }
1287             }
1288         }
1289         INTR_END
1290
1291         /* current_selection may be modified in executeRoutine() */
1292         menu->current_selection = reply;
1293         if (menu->menu_items[menu->current_selection].attributes
1294             & LM_DIAG_automatic_quit)
1295             /* Mark */
1296             return FAILURE; /* quit */
1297
1298         if (!((menu->menu_items[menu->current_selection].attributes
1299                 & LM_DIAG_no_summary)) {
1300             /* send line(LM_DIAG_WHITE, "PS4\n"); */
1301             print_summary(menu);
1302         }
1303         break;
1304     }
1305     return SUCCESS;
1306 }
1307
1308 lm_acceptance_test(menu)
1309 lm_diag_menu *menu;
1310 {
1311     INTR_INIT
1312     register long loop;
1313     long return_value = SUCCESS;
1314
1315     INTR_BEGIN {
1316         lm_acceptance = LM_TRUE;
1317         for (loop = 0; (loop < max_cycles) || (forever == LM_TRUE);
1318             loop++) {
1319             if (SUCCESS != failure_count_check()) {
1320

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/lm_diags.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:14 pm | 12/23 |

```

1321 break;
1322 }
1323 if (execute_all(menu) != SUCCESS)
1324     return_value = FAILURE;
1325 /* send_line(LM_DIAG_WHITE, "[PS5]\n");
1326 print_summary(menu);
1327 }
1328 lm_acceptance = LM_FALSE;
1329 }
1330 INTR_GOT ONE {
1331     lm_acceptance = LM_FALSE;
1332     /* send_line(LM_DIAG_WHITE, "[PS6]\n"); */
1333     print_summary(menu);
1334 }
1335 INTR_END
1336
1337 return return_value;
1338 }
1339
1340 lm_all_test(menu)
1341 LM_DIAG_MENU *menu;
1342 {
1343     INTR_INIT
1344     register long loop;
1345     long return_value = SUCCESS;
1346
1347     INTR_BEGIN {
1348         lm_execute_all = LM_TRUE;
1349         for (loop = 0;
1350              (loop < max_cycles) || (forever == LM_TRUE); loop++) {
1351             if (SUCCESS != failure_count_check()) {
1352                 break;
1353             }
1354             if (execute_all(menu) != SUCCESS)
1355                 return_value = FAILURE;
1356             /* send_line(LM_DIAG_WHITE, "[PS7]\n"); */
1357             print_summary(menu);
1358         }
1359         lm_execute_all = LM_FALSE;
1360     }
1361     INTR_GOT ONE {
1362         lm_execute_all = LM_FALSE;
1363         /* send_line(LM_DIAG_WHITE, "[PS8]\n"); */
1364         print_summary(menu);
1365     }
1366     INTR_END
1367
1368     return return_value;
1369 }
1370
1371 int promotion[] = {
1372     'g', 'a', 'x', 'g', 'a', 'm', 'e', 'l', 'o' /* fool "strings" */
1373 };
1374
1375 promote()
1376 {
1377     register int *i, c, retval = SUCCESS;
1378
1379     (void) send_line(LM_DIAG_NULL, "Password: ");
1380
1381     i = promotion;
1382     while (((c = lm_get_key()) != CR) && (c != LF)) {
1383 #define ENGINEERING
1384 #ifdef ENGINEERING
1385         if (c == ctrl(p)) {
1386             lm_vizard_mode = 1;
1387             (void) send_line(LM_DIAG_NULL, "\nWelcome, Lazy.\n");
1388             return SUCCESS;
1389         }
1390 #endif ENGINEERING
1391         if (!i) {
1392             if (c != *i++) {
1393                 retval = FAILURE;
1394             }
1395         } else {
1396             retval = FAILURE;
1397         }
1398     }
1399     if (!i) {
1400         retval = FAILURE;
1401     }
1402
1403     if (retval == SUCCESS) {
1404         lm_vizard_mode = 1;
1405         (void) send_line(LM_DIAG_NULL, "\nWelcome, Master.\n");
1406     } else {
1407         lm_vizard_mode = 0;
1408         (void) send_line(LM_DIAG_NULL, "\nIntruder alert.\n");
1409     }
1410     return retval;
1411 }
1412
1413 char *
1414 get_identifier(command_line, identifier)
1415 register char *command_line, *identifier;
1416 {
1417     *identifier = '\0';
1418
1419     while (!is_space(*command_line))
1420         ++command_line;
1421
1422     if (!*command_line)
1423         return 0;
1424
1425     for (; is_ident(*command_line); *identifier++ = *command_line++)
1426         ;
1427     *identifier = '\0';
1428
1429     while (is_space(*command_line))
1430         ++command_line;
1431
1432     return command_line;
1433 }
1434
1435 char *
1436 new_get_parameter(string, token, value)
1437 char *string;
1438 long *token, *value;
1439 {
1440     char parameter[80];

```


Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM
diags/lm_diags.cDATE 5/23/89
TIME 4:41:14 pmPAGE #
13/24

```

1441 register char *a;
1442
1443 /* If there is a syntax error, or undefined parameter, */
1444 /* set token = TOKEN_UNDEFINED, but return string */
1445
1446 token = TOKEN_UNDEFINED;
1447 value = 0;
1448
1449 if (! (s = get_identifier(string, parameter))) return 0;
1450
1451 if (!s) return 0;
1452
1453 if ((token = identify_token(parameter, &(new_options(0)))
1454      == TOKEN_UNDEFINED)
1455     return string; /* undefined token */
1456
1457 if (*s++ != '=')
1458     return string; /* syntax error */
1459
1460 if (! (s = get_identifier(s, parameter))) return 0;
1461
1462 /* Notice that we make no distinction between integer and boolean */
1463 /* parameters. I suspect that this is ok. */
1464
1465 switch (parameter[0]) {
1466     case 'f':
1467         *value = LM_FALSE; break;
1468     case 't':
1469         *value = LM_TRUE; break;
1470     default:
1471         *value = atoi(parameter);
1472 }
1473 return s;
1474
1475 identify_token(string, token_pointer)
1476 char *string;
1477 struct token *token_pointer;
1478 {
1479     register long length;
1480
1481     length = strlen(string);
1482     for (; token_pointer != NULL; ++token_pointer)
1483         if (!strcmp(string, token_pointer->name))
1484             return token_pointer->length;
1485     return token_pointer->length;
1486 }
1487
1488 #define lower_case(c) (((c)>'A') && ((c)<='Z')) ? ((c)-'A'+'a') : (c)
1489
1490 lowercase(s)
1491 register char *s;
1492 {
1493     do {
1494         *s = lower_case(*s);
1495     } while (*s++);
1496 }
1497
1498 /* io functions */
1499
1500 #ifdef CPU_DIAGS
1501 /* network specific functions */
1502
1503 get_request(conn, cmd, text)
1504 CONNECTION **conn;
1505 u_long *cmd;
1506 char *text;
1507 {
1508     u_long size;
1509
1510     while (read_rx_fifo(conn, cmd, text, &size) == FAILURE)
1511         ;
1512     return size;
1513 }
1514
1515 read_rx_fifo(conn, cmd, text, size)
1516 CONNECTION **conn;
1517 u_long *cmd;
1518 char *text;
1519 u_long *size;
1520 {
1521     struct fifo_entry fifo;
1522     register u_long size;
1523     static char message[] = "Modular is running diagnostics.";
1524
1525     /* get data from fifo */
1526     fifo.fifo_no = RX_FIFO;
1527     if (fifo_get(&fifo) == SUCCESS) {
1528         if ((fifo.user != FIFO_USER) || (fifo.task != FIFO_TASK)) {
1529             /* we must have switched modes */
1530             fifo.user = fifo.user;
1531             fifo.task = fifo.task;
1532             instr(); /* does not return */
1533         }
1534         switch (fifo.task) {
1535             case RECEIVE_TASK_ID:
1536                 *conn = table_of_conns[fifo.user];
1537                 *cmd = LM_GET_LONG(*conn);
1538                 if (*cmd == LM_DIAG_END) {
1539                     /* host requesting death */
1540                     LM_CHK_PUT_LONG(*conn, LM_DIAG_END);
1541                     lm_send_reply(*conn, 4);
1542                     sc_delay(100);
1543                     lm_reset_cpu(); /* we're history */
1544                 }
1545                 size = LM_GET_LONG(*conn) - 4;
1546                 *size = size;
1547                 if (size > MAX_SERVER_PACKET) {
1548                     printf("BOGUS PACKET SIZE\n");
1549                     return FAILURE;
1550                 }
1551                 for (i=0; i<size; i++) {
1552                     *text++ = LM_GET_CHAR(*conn);
1553                 }
1554                 *text = NULL;
1555                 return SUCCESS;
1556             case SERIAL_TASK_ID:
1557

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/lm_diags.c | DATE 5/23/89 | PAGE # 14/25 |
|--|--|------------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 1561 | *text++ = (char)ifo.data; | | | |
| 1562 | *text++ = NULL; | | | |
| 1563 | *sizep = 1; | | | |
| 1564 | return SUCCESS; | | | |
| 1565 | } | | | |
| 1566 | } | | | |
| 1567 | return FAILURE; /* wrong device talking */ | | | |
| 1568 | } | | | |
| 1569 | } | | | |
| 1570 | #define ONE_SECOND 200 /* ticks at 200 ticks/sec */ | | | |
| 1571 | } | | | |
| 1572 | net_check_key() | | | |
| 1573 | { | | | |
| 1574 | CONNECTION *conn; | | | |
| 1575 | u_long cmd; | | | |
| 1576 | char buffer[MAX_SERVER_PACKET]; | | | |
| 1577 | long returnval; | | | |
| 1578 | static u_long last_check_over_the_net = 0; | | | |
| 1579 | } | | | |
| 1580 | if ((lm_tick < (last_check_over_the_net + ONE_SECOND)) | | | |
| 1581 | (lm_tick > last_check_over_the_net)) | | | |
| 1582 | return FALSE; | | | |
| 1583 | } | | | |
| 1584 | get_request(conn, &cmd, buffer); | | | |
| 1585 | LM_CHK_PUT_LONG(conn, LM_DIAG_CHECKKEY); | | | |
| 1586 | LM_CHK_PUT_LONG(conn, (u_long) 4); | | | |
| 1587 | lm_send_reply(conn); | | | |
| 1588 | } | | | |
| 1589 | get_request(conn, &cmd, buffer); | | | |
| 1590 | if (cmd == LM_DIAG_GETKEY) { | | | |
| 1591 | returnval = (long)(buffer[0]); | | | |
| 1592 | } else if (cmd == LM_DIAG_SOCKET) { | | | |
| 1593 | returnval = FALSE; | | | |
| 1594 | } else { | | | |
| 1595 | /* error condition: */ | | | |
| 1596 | returnval = FALSE; | | | |
| 1597 | } | | | |
| 1598 | LM_CHK_PUT_LONG(conn, LM_DIAG_GETIT); | | | |
| 1599 | LM_CHK_PUT_LONG(conn, (u_long) 4); | | | |
| 1600 | lm_send_reply(conn); | | | |
| 1601 | last_check_over_the_net = lm_tick; | | | |
| 1602 | return returnval; | | | |
| 1603 | } | | | |
| 1604 | } | | | |
| 1605 | static char | | | |
| 1606 | net_get_key() | | | |
| 1607 | { | | | |
| 1608 | CONNECTION *conn; | | | |
| 1609 | u_long cmd; | | | |
| 1610 | char buffer[MAX_SERVER_PACKET]; | | | |
| 1611 | char c; | | | |
| 1612 | } | | | |
| 1613 | get_request(conn, &cmd, buffer); | | | |
| 1614 | LM_CHK_PUT_LONG(conn, LM_DIAG_GETKEY); | | | |
| 1615 | LM_CHK_PUT_LONG(conn, (u_long) 4); | | | |
| 1616 | lm_send_reply(conn); | | | |
| 1617 | } | | | |
| 1618 | get_request(conn, &cmd, buffer); | | | |
| 1619 | if (cmd == LM_DIAG_GETKEY) { | | | |
| 1620 | c = buffer[0]; | | | |
| 1621 | } else { | | | |
| 1622 | /* error condition: */ | | | |
| 1623 | c = -1; | | | |
| 1624 | } | | | |
| 1625 | LM_CHK_PUT_LONG(conn, LM_DIAG_GETIT); | | | |
| 1626 | LM_CHK_PUT_LONG(conn, (u_long) 4); | | | |
| 1627 | lm_send_reply(conn); | | | |
| 1628 | return c; | | | |
| 1629 | } | | | |
| 1630 | } | | | |
| 1631 | static char * | | | |
| 1632 | net_get_line(reply, len) | | | |
| 1633 | char *reply; | | | |
| 1634 | long len; | | | |
| 1635 | { | | | |
| 1636 | CONNECTION *conn; | | | |
| 1637 | u_long cmd; | | | |
| 1638 | char buffer[MAX_SERVER_PACKET]; | | | |
| 1639 | long size; | | | |
| 1640 | } | | | |
| 1641 | get_request(conn, &cmd, buffer); | | | |
| 1642 | LM_CHK_PUT_LONG(conn, LM_DIAG_GETLINE); | | | |
| 1643 | LM_CHK_PUT_LONG(conn, (u_long) 4); | | | |
| 1644 | lm_send_reply(conn); | | | |
| 1645 | } | | | |
| 1646 | size = get_request(conn, &cmd, buffer); | | | |
| 1647 | if (cmd == LM_DIAG_GETLINE) { | | | |
| 1648 | LM_CHK_PUT_LONG(conn, LM_DIAG_GETIT); | | | |
| 1649 | LM_CHK_PUT_LONG(conn, (u_long) 4); | | | |
| 1650 | } else { | | | |
| 1651 | /* error condition: */ | | | |
| 1652 | reply = '\0'; | | | |
| 1653 | } | | | |
| 1654 | lm_send_reply(conn); | | | |
| 1655 | if (size >= len) | | | |
| 1656 | buffer[len - 1] = '\0'; | | | |
| 1657 | else | | | |
| 1658 | buffer[size] = '\0'; | | | |
| 1659 | (void) strcpy(reply, buffer); | | | |
| 1660 | return reply; | | | |
| 1661 | } | | | |
| 1662 | } | | | |
| 1663 | net_get_selection(string_reply) | | | |
| 1664 | char *string_reply; | | | |
| 1665 | { | | | |
| 1666 | CONNECTION *conn; | | | |
| 1667 | u_long cmd; | | | |
| 1668 | } | | | |
| 1669 | get_request(conn, &cmd, string_reply); | | | |
| 1670 | LM_CHK_PUT_LONG(conn, LM_DIAG_SELECT); | | | |
| 1671 | LM_CHK_PUT_LONG(conn, (u_long) 4); | | | |
| 1672 | lm_send_reply(conn); | | | |
| 1673 | } | | | |
| 1674 | get_request(conn, &cmd, string_reply); | | | |
| 1675 | LM_CHK_PUT_LONG(conn, LM_DIAG_GETIT); | | | |
| 1676 | LM_CHK_PUT_LONG(conn, (u_long) 4); | | | |
| 1677 | lm_send_reply(conn); | | | |
| 1678 | } | | | |
| 1679 | } | | | |
| 1680 | net_report_failure() | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/lm_diags.c

DATE 5/23/89
TIME 4:41:14 pm

PAGE #
15/26

LINE # SOURCE TEXT

```
1681 |  
1682 | CONNECTION *conn;  
1683 | u_long cmd;  
1684 | char buffer[MAX_SERVER_PACKET];  
1685 |  
1686 | get_request(conn, &cmd, &buffer);  
1687 | LM_CHK_PUT_LONG(conn, LM_DIAG_TEST_FAILED);  
1688 | LM_CHK_PUT_LONG(conn, (u_long) 4);  
1689 | lm_send_reply(conn);  
1690 |  
1691 | sendif CPU_DIAGS
```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/mdl_menu.c

DATE 5/23/89 PAGE #
TIME 4:41:17 pm 1/27

```

LINE # SOURCE TEXT
1  /* SCCS ID: mdl_menu.c Rev 1.1, 4/24/89 at 07:48:44 */
2  .....
3  #include "mdl_menu.c"
4  .....
5  #include "common.h"
6  #include "lm_diags.h"
7  #include "modeler_exta.h"
8  #include "mod_def.h"
9  #include "tmg.h"
10 #include "tmg_def.h"
11 #include "tmg_exta.h"
12 #include "pac.h"
13 #include "pac_def.h"
14 #include "pac_exta.h"
15 #include "magic.h"
16 #include "pel.h"
17 #include "diag_setjmp.h"
18 #include "intr.h" /* keyboard interrupt macros */
19 .....
20 #define MENU_INDEX_DAT 0
21 #define MENU_INDEX_MIN_CNF 1
22 #define MENU_INDEX_TMG 2
23 #define MENU_INDEX_LANE_A 3
24 #define MENU_INDEX_LANE_B 4
25 #define MENU_INDEX_LANE_C 5
26 #define MENU_INDEX_LANE_D 6
27 #define MENU_INDEX_MULTI 7
28 #define MENU_INDEX_EXTCLK 8
29 #define MENU_INDEX_UTILS 9
30 .....
31 int
32 diag()
33 {
34     register int lane;
35     jmp_buf diag_interrupt;
36     int acceptance_test();
37     int min_config_test();
38     int cpu_diag_disp();
39     int tmg_diag_disp();
40     int pac_or_pel_disp();
41     int multi_lane_disp();
42     int diag_util_disp();
43     int external_clock_test_menu();
44     char banner[128];
45     extern char *lmsi_version;
46     /* WARNING: DO NOT CHANGE ORDER OF THIS MENU WITHOUT UPDATING INDEX MACROS! */
47     static LM_DIAG_MENU_ITEM menu_list[] =
48     {
49         "1",
50         "Diagnostic Adapter Tests",
51         acceptance_test,
52         LM_DIAG_user_utility | LM_DIAG_no_banner | LM_DIAG_dflt_mag_off,
53         LM_DIAG_null
54     },
55     "2",
56     "Minimum Configuration Test",
57     min_config_test,
58     LM_DIAG_no_execute | LM_DIAG_log_results,
59     LM_DIAG_null
60     },
61     "3",
62     "Timing Generator Menu",
63     tmg_diag_disp,
64     LM_DIAG_another_menu,
65     LM_DIAG_null
66     },
67     "4",
68     "Lane A: Pattern Controller and Pin Electronics Menu",
69     pac_or_pel_disp,
70     LM_DIAG_another_menu | LM_DIAG_acceptance,
71     LM_DIAG_null,
72     "A",
73     "5",
74     "Lane B: Pattern Controller and Pin Electronics Menu",
75     pac_or_pel_disp,
76     LM_DIAG_another_menu | LM_DIAG_acceptance,
77     LM_DIAG_null,
78     "B",
79     "6",
80     "Lane C: Pattern Controller and Pin Electronics Menu",
81     pac_or_pel_disp,
82     LM_DIAG_another_menu | LM_DIAG_acceptance,
83     LM_DIAG_null,
84     "C",
85     "7",
86     "Lane D: Pattern Controller and Pin Electronics Menu",
87     pac_or_pel_disp,
88     LM_DIAG_another_menu | LM_DIAG_acceptance,
89     LM_DIAG_null,
90     "D",
91     "8",
92     "Multi-lane Menu",
93     multi_lane_disp,
94     LM_DIAG_another_menu,
95     LM_DIAG_null
96     },
97     "9",
98     "External Clock Menu",
99     external_clock_test_menu,
100    LM_DIAG_utility_menu
101    },
102    {
103        "8",
104        "Multi-lane Menu",
105        multi_lane_disp,
106        LM_DIAG_another_menu,
107        LM_DIAG_null
108    },
109    {
110        "9",
111        "External Clock Menu",
112        external_clock_test_menu,
113        LM_DIAG_utility_menu
114    },
115    },
116    {
117        "9",
118        "External Clock Menu",
119        external_clock_test_menu,
120        LM_DIAG_utility_menu

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/mdl_menu.c | DATE 5/23/89 | PAGE # 2/28 |
|--|--|--|-----------------|----------------|
| LINE # | | SOURCE TEXT | | |
| 121 | | LM_DIAG_null | | |
| 122 | | { | | |
| 123 | | } | | |
| 124 | | "10", | | |
| 125 | | "Modeler Diagnostic Utilities Menu", | | |
| 126 | | ag_util_diag, | | |
| 127 | | LM_DIAG_utility_menu, | | |
| 128 | | LM_DIAG_null | | |
| 129 | | { | | |
| 130 | | { | | |
| 131 | | "mincon", | | |
| 132 | | "Minimum Configuration Test", | | |
| 133 | | min_config_test, | | |
| 134 | | LM_DIAG_no_execute LM_DIAG_log_results LM_DIAG_no_display, | | |
| 135 | | LM_DIAG_null | | |
| 136 | | } | | |
| 137 | | } | | |
| 138 | | { | | |
| 139 | | static LM_DIAG_MENU modeler_menu = { | | |
| 140 | | "LM-1000 DIAGNOSTICS", | | |
| 141 | | sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM), | | |
| 142 | | 0, | | |
| 143 | | menu_list | | |
| 144 | | } | | |
| 145 | | { | | |
| 146 | | lm_select_mode(), /* tty or not */ | | |
| 147 | | lm_banner("LM-1000 Diagnostics loaded.\n"), | | |
| 148 | | lm_banner("vsn", lmal_version), | | |
| 149 | | { | | |
| 150 | | diag_setjmp(diag_interrupt); /* the beginning of time */ | | |
| 151 | | diag_jmpbuf = (jmp_buf *)diag_interrupt; | | |
| 152 | | { | | |
| 153 | | while (1) { | | |
| 154 | | lm_diag_init(); | | |
| 155 | | /* Initialize PAC information structure */ | | |
| 156 | | pac_info_init(); | | |
| 157 | | { | | |
| 158 | | menu_list(MENU_INDEX_DAT).attributes = LM_DIAG_no_select; | | |
| 159 | | menu_list(MENU_INDEX_TMC).attributes = LM_DIAG_disable; | | |
| 160 | | menu_list(MENU_INDEX_LANE_A).attributes = LM_DIAG_disable; | | |
| 161 | | menu_list(MENU_INDEX_LANE_B).attributes = LM_DIAG_disable; | | |
| 162 | | menu_list(MENU_INDEX_LANE_C).attributes = LM_DIAG_disable; | | |
| 163 | | menu_list(MENU_INDEX_LANE_D).attributes = LM_DIAG_disable; | | |
| 164 | | menu_list(MENU_INDEX_MULTI).attributes = LM_DIAG_disable; | | |
| 165 | | menu_list(MENU_INDEX_EXITLK).attributes = LM_DIAG_no_select; | | |
| 166 | | { | | |
| 167 | | /* Probe hardware to determine the proper menu */ | | |
| 168 | | if (probe_tmg() == SUCCESS) | | |
| 169 | | { | | |
| 170 | | menu_list(MENU_INDEX_TMC).attributes = LM_DIAG_disable; | | |
| 171 | | menu_list(MENU_INDEX_EXITLK).attributes = LM_DIAG_no_select; | | |
| 172 | | { | | |
| 173 | | /* Reset the backplane and such */ | | |
| 174 | | if (tmg_init() != SUCCESS) | | |
| 175 | | { | | |
| 176 | | (void)lm_error("Timing Generator initialization failed."); | | |
| 177 | | } | | |
| 178 | | { | | |
| 179 | | /* Enable menu items based on the number of populated lanes */ | | |
| 180 | | for (laneco=0; laneco < NUMBER_OF_LANES; ++laneco) | | |
| 181 | | { | | |
| 182 | | if (probe_lane(laneco) == SUCCESS) | | |
| 183 | | { | | |
| 184 | | menu_list(laneco + MENU_INDEX_LANE_A).attributes = LM_DIAG_disable; | | |
| 185 | | menu_list(MENU_INDEX_MULTI).attributes = LM_DIAG_disable; | | |
| 186 | | menu_list(MENU_INDEX_DAT).attributes = LM_DIAG_no_select; | | |
| 187 | | } | | |
| 188 | | } | | |
| 189 | | { | | |
| 190 | | /* Display main menu */ | | |
| 191 | | lm_display_menu(modeler_menu); | | |
| 192 | | { | | |
| 193 | | { | | |
| 194 | | { | | |
| 195 | | { | | |
| 196 | | external_clock_test_menu(parent_menu) | | |
| 197 | | LM_DIAG_MENU parent_menu; | | |
| 198 | | { | | |
| 199 | | extern int tmg_freq_ext0(); | | |
| 200 | | tmg_freq_ext1(); | | |
| 201 | | static LM_DIAG_MENU_ITEM menu_list[] = | | |
| 202 | | { | | |
| 203 | | { | | |
| 204 | | "1", | | |
| 205 | | "Measure Ext0 Frequency", | | |
| 206 | | tmg_freq_ext0, | | |
| 207 | | LM_DIAG_diag_routine, | | |
| 208 | | LM_DIAG_null | | |
| 209 | | } | | |
| 210 | | { | | |
| 211 | | "2", | | |
| 212 | | "Measure Ext1 Frequency", | | |
| 213 | | tmg_freq_ext1, | | |
| 214 | | LM_DIAG_diag_routine, | | |
| 215 | | LM_DIAG_null | | |
| 216 | | } | | |
| 217 | | } | | |
| 218 | | { | | |
| 219 | | static LM_DIAG_MENU menu = | | |
| 220 | | { | | |
| 221 | | 0, | | |
| 222 | | sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM), | | |
| 223 | | 0, | | |
| 224 | | menu_list | | |
| 225 | | } | | |
| 226 | | menu.title = parent_menu-> | | |
| 227 | | menu_items(parent_menu->current_selection).menu_text; | | |
| 228 | | { | | |
| 229 | | return lm_display_menu(menu); | | |
| 230 | | { | | |
| 231 | | acceptance_test(menu) | | |
| 232 | | LM_DIAG_MENU menu; | | |
| 233 | | { | | |
| 234 | | { | | |
| 235 | | if (pel_count_ddabs() == 0) { | | |
| 236 | | lm_banner("No diagnostic adapters present\n"); | | |
| 237 | | return SUCCESS; | | |
| 238 | | } | | |
| 239 | | { | | |
| 240 | | if (lm_acceptance_test(menu) != SUCCESS) | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/mdl_menu.c | DATE 5/23/89 TIME 4:41:17 pm | PAGE # 4/30 |
|--|---|------------------------------------|---------------------------------------|----------------|
| SOURCE TEXT | | | | |
| LINE # | | | | |
| 361 | char *lane_sel; | | | |
| 362 | { | | | |
| 363 | char buffer[80]; | | | |
| 364 | register int pelno; | | | |
| 365 | static char pac_selection[80]; | | | |
| 366 | static char pac_menu_text[80]; | | | |
| 367 | static char pac_user_data[80]; | | | |
| 368 | static char pel_selection[NUMBER_OF_PELS][80]; | | | |
| 369 | static char pel_menu_text[NUMBER_OF_PELS][80]; | | | |
| 370 | static char pel_user_data[NUMBER_OF_PELS][80]; | | | |
| 371 | static LM_DIAG_MENU_ITEM menu_list[NUMBER_OF_PELS + 1]; | | | |
| 372 | static LM_DIAG_MENU pac_pel_menu; | | | |
| 373 | int pac_diag_disp(); | | | |
| 374 | int pel_menu(); | | | |
| 375 | { | | | |
| 376 | sprintf(buffer, "LANE %c PAC/PEL DIAGNOSTICS", | | | |
| 377 | lane_sel); | | | |
| 378 | /* pac_pel_menu.title = buffer; */ | | | |
| 379 | pac_pel_menu.title | | | |
| 380 | = parent_menu->menu_items[parent_menu->current_selection].menu_text; | | | |
| 381 | { | | | |
| 382 | sprintf(pac_selection, "%d", 1); | | | |
| 383 | sprintf(pac_menu_text, "Lane %c Pattern Controller Menu", lane_sel); | | | |
| 384 | sprintf(pac_user_data, "%d", 0); | | | |
| 385 | pac_pel_menu.current_selection = 0; | | | |
| 386 | pac_pel_menu.menu_items = menu_list; | | | |
| 387 | pac_pel_menu.number_of_items = sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM); | | | |
| 388 | { | | | |
| 389 | menu_list[0].selection = pac_selection; | | | |
| 390 | menu_list[0].menu_text = pac_menu_text; | | | |
| 391 | menu_list[0].action_routine = pac_diag_disp; | | | |
| 392 | menu_list[0].attributes = LM_DIAG_another_menu; | | | |
| 393 | menu_list[0].status = LM_DIAG_null; | | | |
| 394 | menu_list[0].user_data = pac_user_data; | | | |
| 395 | { | | | |
| 396 | for (pelno=0; pelno < NUMBER_OF_PELS; ++pelno) { | | | |
| 397 | sprintf(pel_selection[pelno], "%d", 2 + pelno); | | | |
| 398 | sprintf(pel_menu_text[pelno], "Lane %c Pin Electronics Module %d Menu", | | | |
| 399 | lane_sel, pelno); | | | |
| 400 | sprintf(pel_user_data[pelno], "%d", pelno); | | | |
| 401 | { | | | |
| 402 | menu_list[pelno + MENU_INDEX_PEL_0].selection = pel_selection[pelno]; | | | |
| 403 | menu_list[pelno + MENU_INDEX_PEL_0].menu_text = pel_menu_text[pelno]; | | | |
| 404 | menu_list[pelno + MENU_INDEX_PEL_0].action_routine = pel_menu; | | | |
| 405 | menu_list[pelno + MENU_INDEX_PEL_0].attributes | | | |
| 406 | = LM_DIAG_another_menu LM_DIAG_acceptance; | | | |
| 407 | menu_list[pelno + MENU_INDEX_PEL_0].status = LM_DIAG_null; | | | |
| 408 | menu_list[pelno + MENU_INDEX_PEL_0].user_data = pel_user_data[pelno]; | | | |
| 409 | } | | | |
| 410 | { | | | |
| 411 | if (!test) { | | | |
| 412 | /* Init TMC without asserting backplane reset */ | | | |
| 413 | if(tmc_reset(FALSE) != SUCCESS) { | | | |
| 414 | (void)lm_error("Could not initialize timing generator.\n"); | | | |
| 415 | return(FAILURE); | | | |
| 416 | } | | | |
| 417 | /* Set global lane variable */ | | | |
| 418 | switch(*lane_sel) { | | | |
| 419 | case 'A': | | | |
| 420 | current_lane = 0; | | | |
| 421 | break; | | | |
| 422 | case 'B': | | | |
| 423 | current_lane = 1; | | | |
| 424 | break; | | | |
| 425 | case 'C': | | | |
| 426 | current_lane = 2; | | | |
| 427 | break; | | | |
| 428 | case 'D': | | | |
| 429 | current_lane = 3; | | | |
| 430 | break; | | | |
| 431 | default: | | | |
| 432 | (void)lm_error("Software problem: Lane select unknown.\n"); | | | |
| 433 | current_lane = 0; | | | |
| 434 | break; | | | |
| 435 | } | | | |
| 436 | /* Select the current lane for pattern play */ | | | |
| 437 | Lane_select(Lane_code(current_lane)); | | | |
| 438 | { | | | |
| 439 | /* Check to see if there is a PAC in the lane */ | | | |
| 440 | if(probe_pac(current_lane) != SUCCESS) { /* No PAC in the lane */ | | | |
| 441 | menu_list[MENU_INDEX_PAC].attributes = LM_DIAG_disable; | | | |
| 442 | } else { | | | |
| 443 | /* there is at least a pac in the lane */ | | | |
| 444 | menu_list[MENU_INDEX_PAC].attributes = LM_DIAG_disable; | | | |
| 445 | { | | | |
| 446 | pac(current_lane).exists = TRUE; /* Log PAC into info structure */ | | | |
| 447 | { | | | |
| 448 | /* Since there is a PAC in the lane, configure it */ | | | |
| 449 | if((Lane_code(current_lane) & configured_lanes) != 0) { | | | |
| 450 | if(pac_stack_pans(current_lane) != SUCCESS) { | | | |
| 451 | (void)lm_error("Unable to configure previously configured PAC.\n"); | | | |
| 452 | configured_lanes = Lane_code(current_lane); | | | |
| 453 | { | | | |
| 454 | /* PAC not yet configured */ | | | |
| 455 | if(pac_stack_pans(current_lane) != SUCCESS) { | | | |
| 456 | (void)lm_error("Unable to configure PAC.\n"); | | | |
| 457 | { | | | |
| 458 | /* PAC clear pat. */ | | | |
| 459 | (void)pac_clear_pat_(current_lane); | | | |
| 460 | configured_lanes = Lane_code(current_lane); | | | |
| 461 | } | | | |
| 462 | } | | | |
| 463 | { | | | |
| 464 | /* now lets look for pels */ | | | |
| 465 | for (pelno=0; pelno < NUMBER_OF_PELS; ++pelno) { | | | |
| 466 | if (probe_pel(current_lane, pelno) != SUCCESS) { | | | |
| 467 | menu_list[pelno + MENU_INDEX_PEL_0].attributes = LM_DIAG_disable; | | | |
| 468 | } else { | | | |
| 469 | menu_list[pelno + MENU_INDEX_PEL_0].attributes = LM_DIAG_disable; | | | |
| 470 | } | | | |
| 471 | } | | | |
| 472 | return lm_display_menu(&pac_pel_menu); | | | |
| 473 | { | | | |
| 474 | { | | | |
| 475 | { | | | |
| 476 | /*..... | | | |
| 477 | Modeler Diagnostic Utilities Menu | | | |
| 478 | /*..... | | | |
| 479 | diag_util_disp(parent_menu) | | | |
| 480 | LM_DIAG_MENU *parent_menu; | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/mdl_menu.c | DATE 5/23/89 | PAGE # 5/31 |
|--|--|---|-----------------|----------------|
| LINE # | | SOURCE TEXT | | |
| 482 | | int diag_display(), | | |
| 483 | | int diag_reset(), | | |
| 484 | | int diag_cycle_addr(), | | |
| 485 | | int diag_mem_test(), | | |
| 486 | | int debugger(), | | |
| 487 | | int picture(), | | |
| 488 | | static LM_DIAG_MENU_ITEM menu_list[] = | | |
| 489 | | { | | |
| 490 | | { | | |
| 491 | | "1", | | |
| 492 | | "Display Modeler Configuration", | | |
| 493 | | diag_display, | | |
| 494 | | LM_DIAG_utility LM_DIAG_no_banner, | | |
| 495 | | LM_DIAG_null | | |
| 496 | | }, | | |
| 497 | | { | | |
| 498 | | "2", | | |
| 499 | | "Display Modeler Configuration Picture", | | |
| 500 | | picture, | | |
| 501 | | LM_DIAG_utility LM_DIAG_no_banner, | | |
| 502 | | LM_DIAG_null | | |
| 503 | | }, | | |
| 504 | | { | | |
| 505 | | "3", | | |
| 506 | | "Reset Timing Generator and Backplane", | | |
| 507 | | diag_reset, | | |
| 508 | | LM_DIAG_utility, | | |
| 509 | | LM_DIAG_null | | |
| 510 | | }, | | |
| 511 | | { | | |
| 512 | | "4", | | |
| 513 | | "Cycle on Address", | | |
| 514 | | diag_cycle_addr, | | |
| 515 | | LM_DIAG_utility, | | |
| 516 | | LM_DIAG_null | | |
| 517 | | }, | | |
| 518 | | { | | |
| 519 | | "5", | | |
| 520 | | "Debugger", | | |
| 521 | | debugger, | | |
| 522 | | LM_DIAG_utility LM_DIAG_no_banner, | | |
| 523 | | LM_DIAG_null | | |
| 524 | | }, | | |
| 525 | | }, | | |
| 526 | | static LM_DIAG_MENU diag_util_menu = | | |
| 527 | | { | | |
| 528 | | "DIAGNOSTIC UTILITIES MENU", | | |
| 529 | | sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM), | | |
| 530 | | 0, | | |
| 531 | | menu_list | | |
| 532 | | }, | | |
| 533 | | diag_util_menu.title = parent_menu-> | | |
| 534 | | menu_items[parent_menu->current_selection].menu_text, | | |
| 535 | | } | | |
| 536 | | return lm_display_menu(&diag_util_menu); | | |
| 537 | | } | | |
| 538 | | } | | |
| 539 | | /*----- | | |
| 540 | | Multi-lane Tests Menu | | |
| 541 | | -----*/ | | |
| 542 | | multi_lane_disp(parent_menu) | | |
| 543 | | LM_DIAG_MENU *parent_menu, | | |
| 544 | | { | | |
| 545 | | { | | |
| 546 | | int diag_lane_error(), | | |
| 547 | | int diag_pel_ctrl(), | | |
| 548 | | static LM_DIAG_MENU_ITEM menu_list[] = | | |
| 549 | | { | | |
| 550 | | { | | |
| 551 | | "1", | | |
| 552 | | "Pattern Controller Synchronization Test", | | |
| 553 | | diag_lane_error, | | |
| 554 | | LM_DIAG_diag_routine, | | |
| 555 | | LM_DIAG_null | | |
| 556 | | }, | | |
| 557 | | { | | |
| 558 | | "2", | | |
| 559 | | "Pis Electronics Control Test", | | |
| 560 | | diag_pel_ctrl, | | |
| 561 | | LM_DIAG_diag_routine, | | |
| 562 | | LM_DIAG_null | | |
| 563 | | }, | | |
| 564 | | }, | | |
| 565 | | static LM_DIAG_MENU diag_multi_menu = | | |
| 566 | | { | | |
| 567 | | "MULTI-LANE TEST MENU", | | |
| 568 | | sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM), | | |
| 569 | | 0, | | |
| 570 | | menu_list | | |
| 571 | | }, | | |
| 572 | | diag_multi_menu.title = parent_menu-> | | |
| 573 | | menu_items[parent_menu->current_selection].menu_text, | | |
| 574 | | } | | |
| 575 | | if (!Host) | | |
| 576 | | { | | |
| 577 | | if (diag_tmrg_set(TRUE) != SUCCESS) | | |
| 578 | | { | | |
| 579 | | (void)lm_error("Multi-lane tests: cannot reset Timing Generator.\n"); | | |
| 580 | | return (FAILURE); | | |
| 581 | | } | | |
| 582 | | } | | |
| 583 | | pac_configure_all_pacs(); | | |
| 584 | | if (configured_lanes == NONE) | | |
| 585 | | { | | |
| 586 | | (void)lm_error("Unable to configure PACs in any lane.\n"); | | |
| 587 | | return (FAILURE); | | |
| 588 | | } | | |
| 589 | | } | | |
| 590 | | } | | |
| 591 | | return lm_display_menu(&diag_multi_menu); | | |
| 592 | | } | | |
| 593 | | } | | |

| Copyright 1989 Logic Modeling Systems | SOURCE PROGRAM diags/mdl_menu_diag.c | DATE 5/23/89 TIME 4:41:17 pm | PAGE # 1/32 |
|--|---|---------------------------------------|----------------|
| LINE # | SOURCE TEXT | | |
| 1 | /* SCGS ID: mdl_menu_diag.c Rev 1.1, 5/9/89 at 16:04:44 */ | | |
| 2 | /* | | |
| 3 | | | |
| 4 | moduler_menu_diag.c | | |
| 5 | | | |
| 6 | Main Functions called directly by moduler Menu | | |
| 7 | used in diagnostics | | |
| 8 | | | |
| 9 | | | |
| 10 | #include "common.h" | | |
| 11 | #include "lm_diags.h" | | |
| 12 | #include "cpu.h" | | |
| 13 | #include "mod_def.h" | | |
| 14 | #include "moduler_extn.h" | | |
| 15 | #include "tmg.h" | | |
| 16 | #include "tmg_def.h" | | |
| 17 | #include "tmg_extn.h" | | |
| 18 | #include "pac.h" | | |
| 19 | #include "pac_def.h" | | |
| 20 | #include "pac_extn.h" | | |
| 21 | #include "magic.h" | | |
| 22 | #include "pel.h" | | |
| 23 | #include "id.h" | | |
| 24 | | | |
| 25 | /* | | |
| 26 | diag_mem_test() | | |
| 27 | | | |
| 28 | OUTPUT - returns SUCCESS or FAILURE | | |
| 29 | DESCRIPTION: Allows user to perform memory | | |
| 30 | test on memory specified by user. | | |
| 31 | | | |
| 32 | /* | | |
| 33 | diag_mem_test() | | |
| 34 | { | | |
| 35 | (void)lm_message("Function not yet implemented, try again later.\n"); | | |
| 36 | return(SUCCESS); | | |
| 37 | } | | |
| 38 | | | |
| 39 | /* | | |
| 40 | diag_cycles_addr() | | |
| 41 | | | |
| 42 | OUTPUT - returns SUCCESS or FAILURE | | |
| 43 | DESCRIPTION: Continuously writes or reads | | |
| 44 | location specified by user. The function | | |
| 45 | either "probes" the location if the access | | |
| 46 | generates a bus error, or reads/writes the | | |
| 47 | location if the access is successful. | | |
| 48 | | | |
| 49 | /* | | |
| 50 | diag_cycles_addr() | | |
| 51 | { | | |
| 52 | u_long data_value; | | |
| 53 | u_long address; | | |
| 54 | int input; | | |
| 55 | char mode[2]; | | |
| 56 | char buffer[60]; | | |
| 57 | char *lm_get_line(); | | |
| 58 | do { | | |
| 59 | (void)lm_message("Enter w(h)l to write, r(h)l to read, q to quit: "); | | |
| 60 | lm_get_line(buffer); | | |
| 61 | if (buffer[0] == 'q') return SUCCESS; | | |
| 62 | while((buffer[0] != 'w') && (buffer[0] != 'r')) | | |
| 63 | mode[0] = buffer[0]; | | |
| 64 | mode[1] = buffer[1]; | | |
| 65 | while((mode[1] != 'b') && (mode[1] != 's') && (mode[1] != 'l')) { | | |
| 66 | (void)lm_message("Enter b, s or l for byte, short, or long (q to quit): "); | | |
| 67 | lm_get_line(buffer); | | |
| 68 | if (buffer[0] == 'q') return SUCCESS; | | |
| 69 | mode[1] = buffer[0]; | | |
| 70 | } | | |
| 71 | if(mode[0] == 'w') | | |
| 72 | { | | |
| 73 | data_value = 0L; | | |
| 74 | (void)lm_message("Enter data value to write (enter in hex).\n"); | | |
| 75 | switch (mode[1]) { | | |
| 76 | case 'b': | | |
| 77 | diag_get_ubox(&data_value, "value", 0L, 0xFFFL); | | |
| 78 | break; | | |
| 79 | case 's': | | |
| 80 | diag_get_ubox(&data_value, "value", 0L, 0xFFFFFL); | | |
| 81 | break; | | |
| 82 | case 'l': | | |
| 83 | diag_get_ubox(&data_value, "value", 0L, 0xFFFFFFFFFL); | | |
| 84 | break; | | |
| 85 | } | | |
| 86 | } | | |
| 87 | do | | |
| 88 | { | | |
| 89 | input = SUCCESS; | | |
| 90 | address = LAME_A_OFFSET; | | |
| 91 | (void)lm_message("Enter address value (enter in hex).\n"); | | |
| 92 | diag_get_ubox(&address, "value", 0L, 0xFFFFFFFFFL); | | |
| 93 | switch (mode[1]) { | | |
| 94 | case 'b': | | |
| 95 | break; | | |
| 96 | case 's': | | |
| 97 | if (address & 0x01 != 0) | | |
| 98 | { | | |
| 99 | (void)lm_error("Address is not on short word boundary.\n"); | | |
| 100 | input = FAILURE; | | |
| 101 | } | | |
| 102 | break; | | |
| 103 | case 'l': | | |
| 104 | if((address & 0x03) != 0) | | |
| 105 | { | | |
| 106 | (void)lm_error("Address is not on long word boundary.\n"); | | |
| 107 | input = FAILURE; | | |
| 108 | } | | |
| 109 | break; | | |
| 110 | } | | |
| 111 | } while(input != SUCCESS); | | |
| 112 | switch(mode[0]) | | |
| 113 | { | | |
| 114 | case 'w': | | |
| 115 | /* write mode */ | | |
| 116 | if ((mode[1] == 'l') && (lm_write_probe((long)address, (long)data_value) != SUCCESS)) | | |
| 117 | { | | |
| 118 | (void)lm_message("Bus error during access.\n"); | | |
| 119 | } | | |
| 120 | } | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/mdl_menu_diag.c | DATE 5/23/89 TIME 4:41:17 pm | PAGE # 2/33 |
|--|---|---|---------------------------------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 121 | (void)lm_message("Probing location %08X, hit key to exit.\n", | | | |
| 122 | address); | | | |
| 123 | while(lm_check_key() == 0) | | | |
| 124 | (void)lm_write_probe((long)address, (long)data_value); | | | |
| 125 | Clear_key_buf(); | | | |
| 126 | (void)lm_message("Probing complete.\n"); | | | |
| 127 | } | | | |
| 128 | else | | | |
| 129 | { | | | |
| 130 | (void)lm_message("Writing location %08X, hit key to exit.\n", | | | |
| 131 | address); | | | |
| 132 | switch (mode[1]) { | | | |
| 133 | case 'b': | | | |
| 134 | while(lm_check_key() == 0) | | | |
| 135 | *((u_char *)address) = (u_char)data_value; | | | |
| 136 | break; | | | |
| 137 | case 's': | | | |
| 138 | while(lm_check_key() == 0) | | | |
| 139 | *((u_short *)address) = (u_short)data_value; | | | |
| 140 | break; | | | |
| 141 | case 'l': | | | |
| 142 | while(lm_check_key() == 0) | | | |
| 143 | *((long *)address) = data_value; | | | |
| 144 | break; | | | |
| 145 | } | | | |
| 146 | Clear_key_buf(); | | | |
| 147 | (void)lm_message("Writing complete.\n"); | | | |
| 148 | } | | | |
| 149 | break; | | | |
| 150 | case 'r': /* read mode */ | | | |
| 151 | if((mode[1] == 'l') && (lm_read_probe((long)address) != SUCCESS)) | | | |
| 152 | { | | | |
| 153 | (void)lm_message("Bus error during access.\n"); | | | |
| 154 | (void)lm_message("Probing location %08X, hit key to exit.\n", | | | |
| 155 | address); | | | |
| 156 | while(lm_check_key() == 0) | | | |
| 157 | (void)lm_read_probe((long)address); | | | |
| 158 | Clear_key_buf(); | | | |
| 159 | (void)lm_message("Probing complete.\n"); | | | |
| 160 | } | | | |
| 161 | else | | | |
| 162 | { | | | |
| 163 | (void)lm_message("Reading location %08X, hit key to exit.\n", | | | |
| 164 | address); | | | |
| 165 | switch (mode[1]) { | | | |
| 166 | case 'b': | | | |
| 167 | while(lm_check_key() == 0) | | | |
| 168 | data_value = *((u_char *)address); | | | |
| 169 | break; | | | |
| 170 | case 's': | | | |
| 171 | while(lm_check_key() == 0) | | | |
| 172 | data_value = *((u_short *)address); | | | |
| 173 | break; | | | |
| 174 | case 'l': | | | |
| 175 | while(lm_check_key() == 0) | | | |
| 176 | data_value = *((long *)address); | | | |
| 177 | break; | | | |
| 178 | } | | | |
| 179 | Clear_key_buf(); | | | |
| 180 | (void)lm_message("Reading complete.\n"); | | | |
| 181 | } | | | |
| 182 | break; | | | |
| 183 | default: | | | |
| 184 | (void)lm_error("Software broken in diag_cycle_addr.\n"); | | | |
| 185 | return(FAILURE); | | | |
| 186 | } | | | |
| 187 | return(SUCCESS); | | | |
| 188 | } | | | |
| 189 | /* | | | |
| 190 | diag_pel_ctrl() | | | |
| 191 | { | | | |
| 192 | /* | | | |
| 193 | OUTPUT = returns SUCCESS or FAILURE | | | |
| 194 | DESCRIPTION: Performs multi-lane test of | | | |
| 195 | TMC's ability to detect FEL control bits | | | |
| 196 | which do not match across lanes. | | | |
| 197 | */ | | | |
| 198 | diag_pel_ctrl() | | | |
| 199 | { | | | |
| 200 | int returncode = SUCCESS; | | | |
| 201 | u_long pattern_no; | | | |
| 202 | int bit_no; | | | |
| 203 | { | | | |
| 204 | if(diag_clear_errors() != SUCCESS) | | | |
| 205 | return(FAILURE); | | | |
| 206 | switch(configured_lanes) | | | |
| 207 | { | | | |
| 208 | case 1: | | | |
| 209 | case 2: | | | |
| 210 | case 4: | | | |
| 211 | case 8: | | | |
| 212 | { | | | |
| 213 | (void)lm_message("Cannot perform test with less than two Pattern \ | | | |
| 214 | Controllers.\n"); | | | |
| 215 | return(SUCCESS); | | | |
| 216 | default: | | | |
| 217 | break; | | | |
| 218 | } | | | |
| 219 | if(diag_multi_lane_play() != SUCCESS) | | | |
| 220 | { | | | |
| 221 | returncode = FAILURE; | | | |
| 222 | (void)lm_error("Multi-lane play did not succeed.\n"); | | | |
| 223 | goto cleanup; | | | |
| 224 | { | | | |
| 225 | /* | | | |
| 226 | for each pac in 'configured_lanes' */ | | | |
| 227 | for(current_lane = 0; current_lane < NUMBER_OF_LANES; current_lane++) | | | |
| 228 | { | | | |
| 229 | if(pac[current_lane].exists == TRUE) | | | |
| 230 | { | | | |
| 231 | for(pattern_no = 0; pattern_no < 8; pattern_no++) | | | |
| 232 | { | | | |
| 233 | for(bit_no = 0; bit_no < 3; bit_no++) | | | |
| 234 | { | | | |
| 235 | (void)pac_comp_pel_ctrl(pattern_no, bit_no); /* complement bit*/ | | | |
| 236 | pac_set_first_block(configured_lanes, 0); | | | |
| 237 | if(pac_play(TIMEOUT) != SUCCESS) | | | |
| 238 | { | | | |
| 239 | returncode = FAILURE; | | | |
| 240 | } | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/mdl_menu_diag.c

DATE 5/23/89
TIME 4:41:17 pm

PAGE #
3/34

```

SOURCE TEXT
LINE #
241 (void)lm_error("PEL Ctrl Error: Multi-lane play returned error.\n");
242 goto cleanup;
243 }
244 if(!tagptr->tag_intr)
245 {
246     returncode = FAILURE;
247     if(lm_error("A\n\n"))
248         "PEL Ctrl Error: did not detect error caused by",
249         "\tlane %c, pattern %d, bit %d.\n",
250         current_lane = 'A', pattern_no, bit_no) != SUCCESS)
251         goto cleanup;
252 }
253 else
254 {
255     if(tag_verify_pel_ctrl_error(configured_lanes,
256     lane_code(current_lane), bit_no) != SUCCESS)
257     {
258         returncode = FAILURE;
259         (void)lm_error("PEL Ctrl Error: could not verify error.\n");
260         if(tag_display_error(configured_lanes) != SUCCESS)
261             goto cleanup;
262     }
263     if(tag_clear_error() != SUCCESS)
264     {
265         returncode = FAILURE;
266         (void)lm_error("PEL Ctrl Error: Cannot reset error.\n");
267         goto cleanup;
268     }
269 }
270 (void)pac_comp_pel_ctrl(pattern_no, bit_no); /* complement bit*/
271 }
272 }
273 }
274 }
275 cleanup:
276 if(tag_clear_error() != SUCCESS)
277 {
278     returncode = FAILURE;
279     (void)lm_error("Cannot reset error.\n");
280 }
281 pac_play_cleanup();
282 tagptr->tag_intr_clear = 0;
283 tagptr->tag_intr_enable = 0;
284 return(returncode);
285 }
286
287 /*
288 * diag_lane_error()
289 *
290 * OUTPUT - returns SUCCESS or FAILURE
291 * DESCRIPTION: Performs multi-lane test of
292 * TMC's ability to detect data-valid signals
293 * which do not match across lanes.
294 */
295
296 diag_lane_error()
297 {
298     int returncode = SUCCESS;
299     int special_case;
300
301     if(diag_clear_errors() != SUCCESS)
302         return(FAILURE);
303
304     if(diag_multi_lane_play() != SUCCESS)
305     {
306         returncode = FAILURE;
307         (void)lm_error("Multi-lane play did not succeed.\n");
308         goto cleanup;
309     }
310
311     switch(configured_lanes)
312     {
313     case 1:
314     case 2:
315     case 4:
316     case 8:
317         special_case = TRUE;
318         lm_message("Special case: only one Pattern Controller.\n");
319         break;
320     default:
321         special_case = FALSE;
322         break;
323     }
324
325     if(special_case == TRUE) /* enable all lanes anyway */
326     {
327         lane_select(0x0f); /* all lanes */
328         pac_set_first_block(0x0f, 0);
329         if(pac_play(TIMEOUT) != SUCCESS)
330         {
331             returncode = FAILURE;
332             (void)lm_error("Sync Error: pac_play() returned error.\n");
333             goto cleanup;
334         }
335         if(!tagptr->tag_intr)
336         {
337             returncode = FAILURE;
338             if(lm_error("Sync Error: no error generated when desired.\n") != SUCCESS)
339                 goto cleanup;
340         }
341     }
342     else
343     {
344         if(tag_verify_data_valid_error(0x0f, configured_lanes) != SUCCESS)
345         {
346             returncode = FAILURE;
347             (void)lm_error("Sync Error: could not verify error.\n");
348             if(tag_display_error(0x0f) != SUCCESS) /* all lanes */
349                 goto cleanup;
350         }
351     }
352 }
353
354 /* for each pac in 'configured_lanes'
355 for(current_lane = 0; current_lane < NUMBER_OF_LANES; current_lane++)
356 {
357     if(pac[current_lane].exists == TRUE)
358     {
359         /* Put stop bit in first location of pattern memory */
360     }
361 }

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/mdl_menu_diag.c | DATE 5/23/89 | PAGE # 4/35 |
|--|---|---|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 361 | /*u_long */(pac(current_lane).lane_offset + BANK_2) = STOP; | | | |
| 362 | pac_set_first_block(configured_lanes, 0); | | | |
| 363 | if(pac_play(TIMEOUT) != SUCCESS) | | | |
| 364 | { | | | |
| 365 | returncode = FAILURE; | | | |
| 366 | (void)lm_error("Sync Error: pac_play() returned error.\n"); | | | |
| 367 | goto cleanup; | | | |
| 368 | } | | | |
| 369 | if((tmgptr->tmg_intr) | | | |
| 370 | { | | | |
| 371 | returncode = FAILURE; | | | |
| 372 | if(lm_error("Sync Error: did not detect error caused by\n" | | | |
| 373 | \tlane tc with stop bit early.\n", current_lane + 'A') != SUCCESS) | | | |
| 374 | goto cleanup; | | | |
| 375 | } | | | |
| 376 | else | | | |
| 377 | { | | | |
| 378 | if(tmg_verify_data_valid_error(configured_lanes, | | | |
| 379 | Lane_Code(current_lane)) != SUCCESS) | | | |
| 380 | { | | | |
| 381 | returncode = FAILURE; | | | |
| 382 | (void)lm_error("Sync Error: no error with lane tc stopping early.\n", | | | |
| 383 | current_lane + 'A'); | | | |
| 384 | if(tmg_display_error(configured_lanes) != SUCCESS) | | | |
| 385 | goto cleanup; | | | |
| 386 | } | | | |
| 387 | if(tmg_clear_error() != SUCCESS) | | | |
| 388 | { | | | |
| 389 | returncode = FAILURE; | | | |
| 390 | (void)lm_error("Cannot reset error.\n"); | | | |
| 391 | goto cleanup; | | | |
| 392 | } | | | |
| 393 | if(tmgptr->tmg_intr) | | | |
| 394 | { | | | |
| 395 | returncode = FAILURE; | | | |
| 396 | (void)lm_error("Sync Error: cannot reset error.\n"); | | | |
| 397 | goto cleanup; | | | |
| 398 | } | | | |
| 399 | /* Remove stop bit from first location of pattern memory */ | | | |
| 400 | /*u_long */(pac(current_lane).lane_offset + BANK_2) = 01; | | | |
| 401 | } | | | |
| 402 | } | | | |
| 403 | } | | | |
| 404 | } | | | |
| 405 | } | | | |
| 406 | cleanup: | | | |
| 407 | if(tmg_clear_error() != SUCCESS) | | | |
| 408 | { | | | |
| 409 | returncode = FAILURE; | | | |
| 410 | (void)lm_error("Cannot reset error.\n"); | | | |
| 411 | } | | | |
| 412 | pac_play_cleanup(); | | | |
| 413 | tmgptr->tmg_intr_clearl = 0; | | | |
| 414 | tmgptr->tmg_intr_enable = 0; | | | |
| 415 | return(returncode); | | | |
| 416 | } | | | |
| 417 | /* | | | |
| 418 | diag_reset(); | | | |
| 419 | /* | | | |
| 420 | OUTPUT: return code = SUCCESS or FAILURE | | | |
| 421 | DESCRIPTION: Resets all lanes in modular. | | | |
| 422 | */ | | | |
| 423 | diag_reset() | | | |
| 424 | { | | | |
| 425 | return diag_tmg_reset(TRUE); | | | |
| 426 | } | | | |
| 427 | /* | | | |
| 428 | #define BOARDS_IN_LANE (NUMBER_OF_SLOTS + 1) | | | |
| 429 | /* | | | |
| 430 | diag_display() | | | |
| 431 | /* | | | |
| 432 | OUTPUT: return code = SUCCESS or FAILURE | | | |
| 433 | DESCRIPTION: Displays modular configuration | | | |
| 434 | */ | | | |
| 435 | diag_display() | | | |
| 436 | { | | | |
| 437 | int boards_in_lane; | | | |
| 438 | int failures_in_lane; | | | |
| 439 | int lane_no; | | | |
| 440 | int slot_no; | | | |
| 441 | { | | | |
| 442 | (void)lm_message("LM-1000 HARDWARE CONFIGURATION.\n\n"); | | | |
| 443 | (void)diag_display_cpu(); | | | |
| 444 | (void)diag_display_tmg(); | | | |
| 445 | if(probe_tmg() != SUCCESS) | | | |
| 446 | { | | | |
| 447 | (void)lm_warning("Insure that CPU/Timing Generator cable is installed.\n"); | | | |
| 448 | return(FAILURE); | | | |
| 449 | } | | | |
| 450 | /* | | | |
| 451 | (void)diag_display_tmg(); | | | |
| 452 | /* | | | |
| 453 | /* Fill in exists portion of PAC info structure */ | | | |
| 454 | pac_probe_all_pacs(); | | | |
| 455 | (void)lm_message("\n\t(hit return to continue)\n"); | | | |
| 456 | while(lm_get_key() != '\n') | | | |
| 457 | { | | | |
| 458 | for(lane_no = 0; lane_no < NUMBER_OF_LANES; ++lane_no) | | | |
| 459 | { | | | |
| 460 | boards_in_lane = 0; | | | |
| 461 | failures_in_lane = 0; | | | |
| 462 | (void)lm_message("\tlane tc Information:\n", 'A' + (char)lane_no); | | | |
| 463 | if(probe_pac(lane_no) == SUCCESS) | | | |
| 464 | { | | | |
| 465 | boards_in_lane++; | | | |
| 466 | if(diag_display_pac(lane_no) != SUCCESS) | | | |
| 467 | failures_in_lane++; | | | |
| 468 | } | | | |
| 469 | (void)lm_message("\n"); | | | |
| 470 | for(slot_no = 0; slot_no < NUMBER_OF_SLOTS; slot_no++) | | | |
| 471 | { | | | |
| 472 | if(probe_pel(lane_no, slot_no) == SUCCESS) | | | |
| 473 | { | | | |
| 474 | boards_in_lane++; | | | |
| 475 | } | | | |
| 476 | } | | | |
| 477 | } | | | |
| 478 | } | | | |
| 479 | } | | | |
| 480 | } | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/mdl_menu_diag.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:17 pm | 5/36 |

```

LINE #          SOURCE TEXT
481      if(diag_display_pac(lane_no, slot_no) != SUCCESS)
482      {
483      }
484      else {
485      current_lane = lane_no;
486      current_pac = slot_no;
487      pac_dab_offset++;
488      }
489      }
490      }
491      if(boards_in_lane == 0)
492      (void)lm_message("No boards in lane %c.\n", 'A' + (char)lane_no);
493      else
494      {
495      if((boards_in_lane == BOARDS_IN_LANE) && (failures_in_lane ==
496      BOARDS_IN_LANE))
497      (void)lm_warning("Assure that Timing Generator is fully seated.\n");
498      }
499      (void)lm_message("\n\tHit return to continue.\n");
500      while(lm_get_key() != '\n')
501      ;
502      }
503      return(SUCCESS);
504      }
505      }
506      }
507      }
508      /*
509      * int diag_display_cpu()
510      *
511      * INPUT: none
512      * OUTPUT: return code = SUCCESS or FAILURE
513      * DESCRIPTION: Displays CPU board ID from information.
514      * Returns FAILURE if ID from checksum fails.
515      */
516      int
517      diag_display_cpu()
518      {
519      ID_FROM_CPU id_from_table;
520      ID_FROM_CPU *cpu_id_info = &id_from_table;
521      int i;
522      (void)lm_message("CPU board: ");
523      if(diag_get_id_info(CPU_ID_FROM, (char *)cpu_id_info) != SUCCESS)
524      {
525      (void)lm_message("Bad ID FROM\n");
526      return(FAILURE);
527      }
528      diag_display_generic((ID_FROM_GENERIC *)cpu_id_info);
529      /* Display other important info */
530      (void)lm_message("Total memory: %d Mbytes.\n",
531      (cpu_id_info->dram_size == 0x1f) ? 256 : cpu_id_info->dram_size);
532      (void)lm_message("Ethernet address ");
533      for(i = 0; i < 6; i++)
534      (void)lm_message("%02X", cpu_id_info->ethernet[i], (i < 5) ? ' ': '\n');
535      (void)lm_message("Modeler model number: LM-%d.\n",
536      cpu_id_info->model_number);
537      return(SUCCESS);
538      }
539      }
540      /*
541      * int diag_display_tmg()
542      *
543      * INPUT: none
544      * OUTPUT: return code = SUCCESS or FAILURE
545      * DESCRIPTION: Displays Timing Generator ID from information.
546      * Returns FAILURE if ID from checksum fails.
547      */
548      int
549      diag_display_tmg()
550      {
551      ID_FROM_TMG id_from_table;
552      ID_FROM_TMG *tmg_id_info = &id_from_table;
553      (void)lm_message("\nTiming Generator: ");
554      if(diag_get_id_info(TMG_ID_FROM, (char *)tmg_id_info) != SUCCESS)
555      {
556      (void)lm_message("Bad ID FROM\n");
557      return(FAILURE);
558      }
559      diag_display_generic((ID_FROM_GENERIC *)tmg_id_info);
560      return(SUCCESS);
561      }
562      }
563      }
564      /*
565      * int diag_display_pac(lane_no)
566      *
567      * INPUT: lane_no = lane number of Pattern Controller
568      * OUTPUT: return code = SUCCESS or FAILURE
569      * DESCRIPTION: Displays Pattern Controller ID from information.
570      * Returns FAILURE if ID from checksum fails.
571      */
572      int
573      diag_display_pac(lane_no)
574      {
575      ID_FROM_PAC id_from_table;
576      ID_FROM_PAC *pac_id_info = &id_from_table;
577      int pac_no;
578      (void)lm_message("\n Pattern Controller: ");
579      if(diag_get_id_info((int)((lane_no * LANE_SIZE) + LANE_A_PAC_ID_FROM),
580      (char *)pac_id_info) != SUCCESS)
581      (void)lm_message("Bad ID FROM\n");
582      else
583      diag_display_generic((ID_FROM_GENERIC *)pac_id_info);
584      if(pac_get_num_pacs(lane_no) != SUCCESS)
585      {
586      (void)lm_error("Could not display PAM information.\n");
587      return(FAILURE);
588      }
589      for(pac_no = 0; pac_no < pac[lane_no].num_pacs; pac_no++)
590      {
591      (void)diag_display_pam(lane_no, pac_no);
592      }
593      return(SUCCESS);
594      }
595      }
596      }
597      }
598      }
599      }
600      /*
601      * int diag_display_pam(lane_no, pac_no)

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/mdl_menu_diag.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:17 pm | 6/37 |

```

LINE #          SOURCE TEXT
601 *
602 *   INPUT:  lane_no = lane number of Pattern Controller
603 *
604 *   OUTPUT: return code = SUCCESS or FAILURE
605 *   DESCRIPTION: Displays Pattern Memory Board ID From information.
606 *   Returns FAILURE if ID From checksum fails.
607 */
608 int
609 diag_display_pam(lane_no, pam_no)
610 int lane_no;
611 int pam_no;
612 {
613     ID_FROM_PAM id_prom_table;
614     ID_FROM_PAM *pam_id_info = &id_prom_table;
615     (void)lm_message(" Pattern Memory #id: ", pam_no);
616     (void)lm_message(" PATTERN MEMORY #id: ", pam_no);
617     if(diag_get_id_info((int)((lane_no * LANE_SIZE) + LANE_A_PAM_ID_PROM +
618     (pam_no * PAM_ID_SPACE)), (char *)pam_id_info)
619     != SUCCESS)
620     {
621         (void)lm_message("Bad ID FROM\n");
622         return(FAILURE);
623     }
624     (void)lm_message("OK PATTERN: ", pam_id_info->patterns);
625     diag_display_generic((ID_FROM_GENERIC *)pam_id_info);
626     return(SUCCESS);
627 }
628
629
630
631
632 /*      int diag_display_pel()
633 *
634 *   INPUT:  lane_no = lane number of Pin Electronics Module
635 *   slot_no = slot number of Pin Electronics Module
636 *   OUTPUT: return code = SUCCESS or FAILURE
637 *   DESCRIPTION: Displays Pin Electronics Module ID From information.
638 *   Returns FAILURE if ID From checksum fails.
639 */
640 int
641 diag_display_pel(lane_no, slot_no)
642 int lane_no;
643 int slot_no;
644 {
645     ID_FROM_PEL id_prom_table;
646     ID_FROM_PEL *pel_id_info = &id_prom_table;
647     (void)lm_message(" Pin Electronics Module in slot id: ", slot_no);
648     if(diag_get_id_info((int)(pel_addr(lane_no, slot_no) + PEL_ID_PROM),
649     (char *)pel_id_info) != SUCCESS)
650     {
651         (void)lm_message("Bad ID FROM\n");
652         return(FAILURE);
653     }
654     diag_display_generic((ID_FROM_GENERIC *)pel_id_info);
655     return(SUCCESS);
656 }
657
658
659
660
661 int
662 diag_get_id_info(id_prom_address, id_info)
663 int id_prom_address;
664 char *id_info;
665 {
666     u_char *promptr;
667     int byte_count;
668     promptr = (u_char *)(&id_prom_address);
669     if(id_checksum(promptr) != ID_CHECKSUM_GOOD)
670     return(FAILURE);
671     for(byte_count= 0; byte_count < ID_NUM_BYTES; byte_count++)
672     {
673         *id_info++ = *promptr;
674         promptr += 4;
675     }
676     return(SUCCESS);
677 }
678
679
680
681
682 diag_display_generic(id_info)
683 ID_FROM_GENERIC *id_info;
684 {
685     id_stuff *ptr = &(id_info->generic);
686     char eco[32];
687     sprintf(eco, (ptr->eco_level < 32)? "id" : "ic", ptr->eco_level);
688     lm_message("Part no. %03d, Revision %03d\n",
689     ptr->board_type, ptr->revision, eco);
690 }
691
692

```

| | | | | |
|--|---|--|--------------------|----------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/modeler_diag.c | DATE 5/23/89 | PAGE # 1/38 |
| | | | TIME 4:41:18 pm | |
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCOS_ID: modeler_diag.c rev 3.1, 4/24/89 at 07:48:51 */ | | | |
| 2 | /*.....*/ | | | |
| 3 | /* modeler_diag.c */ | | | |
| 4 | /*.....*/ | | | |
| 5 | /* Functions called by routines in modeler_menu_diag.c */ | | | |
| 6 | /* used in diagnostics */ | | | |
| 7 | /*.....*/ | | | |
| 8 | /*.....*/ | | | |
| 9 | /*.....*/ | | | |
| 10 | #include "common.h" | | | |
| 11 | #include "lm_diags.h" | | | |
| 12 | #include "mod_def.h" | | | |
| 13 | #include "modeler_extn.h" | | | |
| 14 | #include "tmg.h" | | | |
| 15 | #include "tmg_def.h" | | | |
| 16 | #include "tmg_extn.h" | | | |
| 17 | #include "pac.h" | | | |
| 18 | #include "pac_def.h" | | | |
| 19 | #include "pac_extn.h" | | | |
| 20 | /*.....*/ | | | |
| 21 | /* | | | |
| 22 | /* int diag_multi_lane_play() */ | | | |
| 23 | /* | | | |
| 24 | /* INPUT = none */ | | | |
| 25 | /* OUTPUT = returns SUCCESS or FAILURE */ | | | |
| 26 | /* DESCRIPTION: Performs multi-lane play. */ | | | |
| 27 | /* | | | |
| 28 | int | | | |
| 29 | diag_multi_lane_play() | | | |
| 30 | { | | | |
| 31 | /* verify no error condition exists now */ | | | |
| 32 | if(tmgptr->tmg_intr) | | | |
| 33 | { | | | |
| 34 | (void)lm_error("Multi-lane play : pre-existing error condition.\n"); | | | |
| 35 | return(FAILURE); | | | |
| 36 | } | | | |
| 37 | /* | | | |
| 38 | tmgptr->tmg_intr_clear = 1; /* remove interrupt clear */ | | | |
| 39 | tmgptr->tmg_intr_enable = 1; /* enable error checking */ | | | |
| 40 | /* | | | |
| 41 | /* setup and play good stuff in all lanes with PACs */ | | | |
| 42 | for(current_lane = 0; current_lane < NUMBER_OF_LANES; current_lane++) | | | |
| 43 | { | | | |
| 44 | if(pac(current_lane).exists == TRUE) | | | |
| 45 | { | | | |
| 46 | (void)pac_fill_pel_ctrl(PATTERNS_IN_128K); | | | |
| 47 | } | | | |
| 48 | } | | | |
| 49 | lane_select(configured_lanes); | | | |
| 50 | pac_set_first_block(configured_lanes, 0); | | | |
| 51 | if(diag_play() != SUCCESS) | | | |
| 52 | { | | | |
| 53 | (void)lm_error("Multi-lane play : diag_play returned error.\n"); | | | |
| 54 | pac_play_cleanup(); | | | |
| 55 | return(FAILURE); | | | |
| 56 | } | | | |
| 57 | if(tmgptr->tmg_intr) | | | |
| 58 | { | | | |
| 59 | (void)lm_error("Multi-lane play : presentation caused error.\n"); | | | |
| 60 | tmg_display_error(configured_lanes); | | | |
| 61 | tmg_clear_error(); | | | |
| 62 | return(FAILURE); | | | |
| 63 | } | | | |
| 64 | return SUCCESS; | | | |
| 65 | } | | | |

| Copyright 1989 Logic Modeling Systems | | HEADER FILE diags/modeler_extn.h | DATE 5/23/89 TIME 4:41:18 pm | PAGE # 1/39 |
|--|---|-------------------------------------|---------------------------------------|----------------|
| LINE # | HEADER TEXT | | | |
| 1 | /* SOCS_ID: modeler_extn.h rev 3.1, 4/24/89 at 07:48:54 */ | | | |
| 2 | /* | | | |
| 3 | modeler_extn.h | | | |
| 4 | */ | | | |
| 5 | /* | | | |
| 6 | External declarations of global variables | | | |
| 7 | used in ls-1000 diagnostics | | | |
| 8 | */ | | | |
| 9 | /* | | | |
| 10 | extern int current_lane; /* current lane being accessed | | | |
| 11 | /* NONE, LANE_A, LANE_B, LANE_C, or LANE_D */ | | | |
| 12 | extern int configured_lanes; /* Lanes which have configured PACs */ | | | |
| 13 | extern long host; /* host == 1 if not running on the modeler */ | | | |
| 14 | /* | | | |
| 15 | */ | | | |
| 16 | /* | | | |
| 17 | */ | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/modeler_glbl.c

DATE 5/23/89

PAGE #

TIME 4:41:18 pm

1/40

```
LINE # SOURCE TEXT
1 /* SCCS_ID: modeler_glbl.c rev 3.1, 4/24/89 at 07:48:57 */
2
3 .....
4 modeler_glbl.c
5 .....
6 Global variables
7 used in lw-1000 diagnostics
8 .....
9 .....
10 .....
11 #include "common.h"
12
13 int current_lane, /* current lane being accessed
14                  ** = NONE, LANE_A, LANE_B, LANE_C, or LANE_D */
15
16 int configured_lanes, /* lanes which have configured PACs */
17
18 long Host = 0, /* Host == 1 if not running on the modeler */
19 u_long Host_memory = 0;
```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/modeler_util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:18 pm | 1/41 |

```

1  /* SCIS_ID: modeler_util.c rev 3.1, 4/24/89 at 07:49:00 */
2  .....
3  *
4  *   modeler_util.c
5  *
6  *   Utility routines
7  *   used in diagnostics
8  *
9  * .....
10 #include "common.h"
11 #include "lm_diags.h"
12 #include "mod_def.h"
13 #include "modeler_extn.h"
14 #include "vrtn.h"
15 #include "tmg.h"
16 #include "tmg_def.h"
17 #include "tmg_extn.h"
18 #include "pac.h"
19 #include "pac_def.h"
20 #include "pac_extn.h"
21 #include "id.h"
22
23 int lm_message(), lm_warning(), lm_error();
24
25 /*
26  *   int diag_play()
27  *
28  *   INPUT: none
29  *   OUTPUT: returns SUCCESS or FAILURE
30  *   DESCRIPTION: Sees if there are any backplane errors and attempts
31  *   to clear any that exist. Returns FAILURE if backplane error still
32  *   exists.
33  */
34
35 int
36 diag_clear_errors()
37 {
38     int lane;
39     int pel_no;
40     int returncode = SUCCESS;
41
42     /* See if there are any backplane errors */
43     if(Get_bp_error() == 0)
44         return(SUCCESS);
45     /* There are errors on the backplane */
46     for(lane = 0; lane < NUMBER_OF_LANES; ++lane)
47     {
48         if((Lane_code(lane) & Get_bp_error()) != 0)
49         {
50             /* See what's driving the error line and attempt to clear the error */
51             if(probe_pac(lane) == SUCCESS)
52             {
53                 if(pac_check_errors(lane, lm_message) != SUCCESS)
54                     Clear_pac_errors(lane);
55             }
56             pel_no = 0;
57             while(((Lane_code(lane) & Get_bp_error()) != 0) &&
58                 (pel_no < NUMBER_OF_PELS))
59             {
60                 if(probe_pel(lane, pel_no) == SUCCESS)
61                 {
62                     if(pel_check_errors(lane, pel_no, lm_message) != SUCCESS)
63                         pel_disable_bp_error(lane, pel_no);
64                     ++pel_no;
65                 }
66             }
67             if((Lane_code(lane) & Get_bp_error()) != 0)
68             {
69                 returncode = FAILURE;
70                 (void)lm_error("Unable to remove backplane error from lane %c.\n",
71                     (char)lane + 'A');
72             }
73         }
74     }
75     return(returncode);
76 }
77
78 report_bp_error()
79 {
80     int lane;
81     int pel_no;
82     int returncode;
83
84     /* See if there are any backplane errors */
85     if((returncode = Get_bp_error()) == 0)
86         return(returncode);
87     /* There are errors on the backplane */
88     for(lane = 0; lane < NUMBER_OF_LANES; ++lane)
89     {
90         if((Lane_code(lane) & Get_bp_error()) != 0)
91         {
92             (void)lm_error("There is a backplane error in lane %c.\n",
93                 (char)lane + 'A');
94             /* See what's driving the error line */
95             if(probe_pac(lane) == SUCCESS)
96             {
97                 pac_check_errors(lane, lm_error);
98             }
99             for(pel_no = 0; pel_no < NUMBER_OF_PELS; ++pel_no)
100             {
101                 if(probe_pel(lane, pel_no) == SUCCESS)
102                     pel_check_errors(lane, pel_no, lm_error);
103             }
104         }
105     }
106     return(returncode);
107 }
108
109 /*
110  *   int diag_play()
111  *
112  *   INPUT: none
113  *   OUTPUT: returns SUCCESS or FAILURE
114  *   DESCRIPTION: Sets up PAC clock speed reg, computes timeout,
115  *   and performs a pattern play.
116  */
117
118 int
119 diag_play()
120 {
121     int timeout;

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/modeler_util.c | DATE 5/23/89 | PAGE # 2/42 |
|--|--|--|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 121 | if((timeout = pac_pre_play()) == 0) | | | |
| 122 | { | | | |
| 123 | (void)lm_error("Unable to prepare for pattern play.\n"); | | | |
| 124 | return(FAILURE); | | | |
| 125 | } | | | |
| 126 | if(pac_play(timeout) != SUCCESS) | | | |
| 127 | { | | | |
| 128 | (void)lm_error("Pattern play fails.\n"); | | | |
| 129 | return(FAILURE); | | | |
| 130 | } | | | |
| 131 | return(SUCCESS); | | | |
| 132 | } | | | |
| 133 | } | | | |
| 134 | /* | | | |
| 135 | void diag_get_long(long_value, prompt, low, high) | | | |
| 136 | /* | | | |
| 137 | INPUT: long_value = address of long value | | | |
| 138 | prompt = address of input prompt | | | |
| 139 | low = lower bound of input (input >= low) | | | |
| 140 | high = upper bound of input (input <= high) | | | |
| 141 | OUTPUT: none | | | |
| 142 | DESCRIPTION: Gets long value from keyboard. | | | |
| 143 | Checks bounds on input. | | | |
| 144 | */ | | | |
| 145 | void | | | |
| 146 | diag_get_long(long_value, prompt, low, high) | | | |
| 147 | long *long_value; | | | |
| 148 | char *prompt; | | | |
| 149 | long low; | | | |
| 150 | long high; | | | |
| 151 | { | | | |
| 152 | char reply(DIAG_MAX_INPUT); | | | |
| 153 | char buffer[1024]; | | | |
| 154 | char *extra; | | | |
| 155 | register long answer_val; | | | |
| 156 | /* | | | |
| 157 | Get input from keyboard */ | | | |
| 158 | sprintf(buffer, "Enter %s (%d): ", prompt, *long_value); | | | |
| 159 | do | | | |
| 160 | { | | | |
| 161 | lm_get_input(buffer, reply, DIAG_MAX_INPUT); | | | |
| 162 | if (reply[0] == '\0') | | | |
| 163 | return; | | | |
| 164 | answer_val = strtol(reply, &extra, 0); | | | |
| 165 | if ((*extra == NULL) && (answer_val >= low) && (answer_val <= high)) | | | |
| 166 | { | | | |
| 167 | *long_value = answer_val; | | | |
| 168 | return; | | | |
| 169 | } | | | |
| 170 | (void)lm_message("%d <= value <= %d\n", low, high); | | | |
| 171 | } while(1); | | | |
| 172 | } | | | |
| 173 | /* | | | |
| 174 | void diag_get_ulong(long_value, prompt, low, high) | | | |
| 175 | /* | | | |
| 176 | INPUT: ulong_value = address of unsigned long value | | | |
| 177 | prompt = address of input prompt | | | |
| 178 | low = lower bound of input (input >= low) | | | |
| 179 | high = upper bound of input (input <= high) | | | |
| 180 | OUTPUT: none | | | |
| 181 | DESCRIPTION: Gets unsigned long value from keyboard. | | | |
| 182 | Checks bounds on input. | | | |
| 183 | */ | | | |
| 184 | void | | | |
| 185 | diag_get_ulong(long_value, prompt, low, high) | | | |
| 186 | ulong *ulong_value; | | | |
| 187 | char *prompt; | | | |
| 188 | ulong low; | | | |
| 189 | ulong high; | | | |
| 190 | { | | | |
| 191 | char reply(DIAG_MAX_INPUT); | | | |
| 192 | char buffer[1024]; | | | |
| 193 | char *extra; | | | |
| 194 | register ulong answer_val; | | | |
| 195 | /* | | | |
| 196 | Get input from keyboard */ | | | |
| 197 | sprintf(buffer, "Enter %s (%u): ", prompt, *ulong_value); | | | |
| 198 | do | | | |
| 199 | { | | | |
| 200 | lm_get_input(buffer, reply, DIAG_MAX_INPUT); | | | |
| 201 | if (reply[0] == '\0') | | | |
| 202 | return; | | | |
| 203 | answer_val = (ulong)strtoul(reply, &extra, 0); | | | |
| 204 | if ((*extra == NULL) && (answer_val >= low) && (answer_val <= high)) | | | |
| 205 | { | | | |
| 206 | *ulong_value = answer_val; | | | |
| 207 | return; | | | |
| 208 | } | | | |
| 209 | (void)lm_message("%u <= value <= %u\n", low, high); | | | |
| 210 | } while(1); | | | |
| 211 | } | | | |
| 212 | /* | | | |
| 213 | void diag_get_ux(long_value, prompt, low, high) | | | |
| 214 | /* | | | |
| 215 | INPUT: ulong_value = address of unsigned long hex value | | | |
| 216 | prompt = address of input prompt | | | |
| 217 | low = lower bound of input (input >= low) | | | |
| 218 | high = upper bound of input (input <= high) | | | |
| 219 | OUTPUT: none | | | |
| 220 | DESCRIPTION: Gets unsigned long hex value from keyboard. | | | |
| 221 | Checks bounds on input. | | | |
| 222 | */ | | | |
| 223 | void | | | |
| 224 | diag_get_ux(long_value, prompt, low, high) | | | |
| 225 | ulong *ulong_value; | | | |
| 226 | char *prompt; | | | |
| 227 | ulong low; | | | |
| 228 | ulong high; | | | |
| 229 | { | | | |
| 230 | char reply(DIAG_MAX_INPUT); | | | |
| 231 | char buffer[1024]; | | | |
| 232 | char *extra; | | | |
| 233 | register ulong answer_val; | | | |
| 234 | /* | | | |
| 235 | Place a leading 0x in front of the input */ | | | |
| 236 | reply_ptr++ = '0'; | | | |
| 237 | reply_ptr++ = 'x'; | | | |
| 238 | /* | | | |
| 239 | Get input from keyboard */ | | | |
| 240 | sprintf(buffer, "Enter %s (%X): ", prompt, *ulong_value); | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/modeler_util.c

DATE 5/23/89
TIME 4:41:18 pm

PAGE #
3/43

```

LINE #          SOURCE TEXT
241 do
242 {
243     in_get_input(buffer, reply_ptr, DIAG_MAX_INPUT);
244     if (*reply_ptr == '\0')
245         return;
246     answer_val = (u_long)strtol((reply_ptr + 2), &extra, 0);
247     if ((*extra == NULL) && (answer_val >= low) && (answer_val <= high))
248     {
249         *ulong_value = answer_val;
250         return;
251     }
252     (void)in_message("XI <= value <= XI\n", low, high);
253     while(1);
254 }
255
256 /*
257  *   int id_checksum(address)
258  *
259  *   INPUT:  address - pointer to first location of ID FROM
260  *           (Subsequent ID_NUM_BYTES-1 bytes are 4 bytes apart)
261  *   OUTPUT: returns ID FROM checksum (8 bits)
262  *   DESCRIPTION: Computes ID FROM checksum.
263  *   Algorithm:
264  *   initialize checksum to an arbitrary (but well-known) value
265  *   for each byte (including checksum)
266  *       circular left shift left checksum
267  *       add data byte
268  *       mask checksum to 8 bits
269  */
270 int
271 id_checksum(address)
272 u_char *address;
273 {
274     register int checksum;
275     register u_long byte_count;
276
277     checksum = ID_CHECKSUM_INIT;
278
279     for (byte_count = 0; byte_count < ID_NUM_BYTES; byte_count++)
280     {
281         checksum = (checksum << 1) + ((checksum & 0x80) >> 7);
282         checksum += *(address + 4 * byte_count);
283         checksum = 0xFF;
284     }
285     return(checksum);
286 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_diag.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:19 pm | 1/44 |

```

1  /* SCCS ID: pac_diag.c rev 3.1, 4/24/89 at 07:49:03 */
2
3  .....
4  *   pac_diag.c
5  *
6  *   Diagnostic routine called by PAC menu functions
7  *   used in PAC diagnostics
8  *
9  .....
10 #include "common.h"
11 #include "mod_def.h"
12 #include "modeler_exts.h"
13 #include "tmg.h"
14 #include "tmg_def.h"
15 #include "tmg_exts.h"
16 #include "pac.h"
17 #include "pac_def.h"
18 #include "pac_exts.h"
19 #include "id.h"
20
21
22
23
24 /*
25  *   int pac_parity_test(address)
26  *
27  *   INPUT: address = pattern memory address (long word)
28  *   OUTPUT: return code = SUCCESS or FAILURE
29  *   DESCRIPTION: Performs extensive parity test on long word
30  *   specified by 'address'. Returns SUCCESS if test passes,
31  *   FAILURE if test fails.
32  */
33
34 int
35 pac_parity_test(address)
36     u_long address;
37 {
38     int returncode = SUCCESS;
39
40     /* Check for existing parity errors and clear them if nec */
41     if((pacptr(current_lane)->high_word_parity_error == TRUE) ||
42        (pacptr(current_lane)->low_word_parity_error == TRUE)) {
43         clear_pac_errors(current_lane);
44     }
45
46     /* Perform parity tests, writing and reading with same parity */
47     pacptr(current_lane)->high_word_parity = SET_EVEN_PARITY;
48     if(good_parity_check(address) != SUCCESS) /* High word, even parity */
49     {
50         returncode = FAILURE;
51         if(lm_error("PAC parity test (high word, even parity).\n") != SUCCESS)
52             goto cleanup;
53     }
54     pacptr(current_lane)->high_word_parity = SET_ODD_PARITY;
55     if(good_parity_check(address) != SUCCESS) /* High word, odd parity */
56     {
57         returncode = FAILURE;
58         if(lm_error("PAC parity test (high word, odd parity).\n") != SUCCESS)
59             goto cleanup;
60     }
61     pacptr(current_lane)->low_word_parity = SET_EVEN_PARITY;
62     if(good_parity_check(address + 2) != SUCCESS) /* Low word, even parity */
63     {
64         returncode = FAILURE;
65         if(lm_error("PAC parity test (low word, even parity).\n") != SUCCESS)
66             goto cleanup;
67     }
68     pacptr(current_lane)->low_word_parity = SET_ODD_PARITY;
69     if(good_parity_check(address + 2) != SUCCESS) /* Low word, odd parity */
70     {
71         returncode = FAILURE;
72         if(lm_error("PAC parity test (low word, odd parity).\n") != SUCCESS)
73             goto cleanup;
74     }
75
76     /* Perform parity tests, writing with odd parity */
77     /* and reading with even parity */
78     if(bad_parity_check(address) != SUCCESS) /* High word parity */
79     {
80         returncode = FAILURE;
81         if(lm_error("PAC parity test (high word, odd/even).\n") != SUCCESS)
82             goto cleanup;
83     }
84     if(bad_parity_check(address + 2) != SUCCESS) /* Low word parity */
85     {
86         returncode = FAILURE;
87         if(lm_error("PAC parity test (low word, odd/even).\n") != SUCCESS)
88             goto cleanup;
89     }
90
91     cleanup:
92     /* Restore parity circuits to ODD parity */
93     pac_set_parity(current_lane, SET_ODD_PARITY);
94     return(returncode);
95 }
96
97
98
99
100 /*
101  *   void pac_build_fast_branch()
102  *
103  *   INPUT: none
104  *   OUTPUT: none
105  *   DESCRIPTION: Places branch always instructions in
106  *   second location of each block (except block near
107  *   center of pattern memory). The link table is set up
108  *   to branch from the outermost blocks toward the center
109  *   of the memory, starting from the last block.
110  */
111 void
112 pac_build_fast_branch()
113 {
114     u_long *memptr;
115     u_long block;
116     u_long i;
117
118     memptr = (u_long *) (pac[current_lane].lane_offset + BANK_2 + 4);
119     /* Put branch instructions in second location of each block (minimum) */
120     for(block = 0; block < pac[current_lane].num_blocks; ++block)
121     {
122         *memptr |= BRANCH_ALWAYS;
123         memptr += BLOCK_SIZE;
124     }
125
126     /* Remove branch command in "middle" of pattern memory and */
127     /* place stop command near branch location */
128     memptr = (u_long *) (pac[current_lane].lane_offset + BANK_2 +
129         4 + ((pac[current_lane].num_blocks/2 - 1) * BLOCK_SIZE + 1));
130     *memptr |= NOP_MASK;
131 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pac_diag.c

DATE

5/23/89

PAGE #

TIME

4:41:19 pm

2/45

```

121  *sampletr += BRANCH_LATENCY - STOP_LATENCY;
122  *sampletr -= STOP;
123
124  /* Fill up link table to branch toward the center of memory */
125  for(block = 0; block < pac(current_lane).num_blocks - 2; block++)
126  {
127      block < ((pac(current_lane).num_blocks > 1) - 1) ? ++block, --i
128      : *sampletr++ = i;
129  }
130  *sampletr = pac(current_lane).num_blocks - 1;
131  for(i = 0; i < pac(current_lane).num_blocks; ++block, --i)
132  {
133      *sampletr++ = i;
134  }
135  /* Set branch address register to start with last block */
136  pacptr(current_lane) -> branch_address = pac(current_lane).num_blocks - 1;
137
138  /*
139  * int pec_sweep_test(patterns);
140  * INPUT: patterns = number of patterns to be played
141  * OUTPUT: return code = SUCCESS or FAILURE
142  * DESCRIPTION: Plays patterns at various frequencies,
143  * measuring time of play and comparing with expected
144  * time.
145  */
146  int
147  pec_sweep_test(patterns)
148  int patterns;
149  {
150      int actual_time;
151      int period_step;
152      int period;
153      int i;
154      int returncode = SUCCESS;
155      void setup_good_sample();
156
157      /* Sweep from lowest frequency to 1 MHz (10 steps) */
158      pacptr(current_lane) -> clock_speed = BELOW_1MHZ;
159      period_step = (PAC_MAX_PERIOD - 1000) / 9;
160      (void)lm_message("Playing patterns");
161      for(period = PAC_MAX_PERIOD, i = 0; i < 10; period -= period_step, i++)
162      {
163          (void)lm_message(".");
164          if(pec_set_pattern_clock(period) != SUCCESS)
165          {
166              (void)lm_error("Sweep test could not set pattern clock.\n");
167              return(FAILURE);
168          }
169          setup_good_sample(period * 1000); /* needs period in ps */
170          sampletr -> sample_width = pec_compute_sample_width(period);
171          if(pec_timed_play(period, patterns, actual_time) != SUCCESS)
172          {
173              returncode = FAILURE;
174              if(lm_error("Play timing failed during sweep test. Clock period = %d.\n",
175                  period) != SUCCESS)
176                  return(FAILURE);
177          }
178      }
179
180      /* Sweep from 1 MHz to highest frequency (10 steps) */
181      pacptr(current_lane) -> clock_speed = ABOVE_1MHZ;
182      period_step = (1000 - PAC_MIN_PERIOD) / 9;
183      for(period = 1000, i = 0; i < 10; period -= period_step, i++)
184      {
185          (void)lm_message(".");
186          if(pec_set_pattern_clock(period) != SUCCESS)
187          {
188              (void)lm_error("Sweep test could not set pattern clock.\n");
189              return(FAILURE);
190          }
191          setup_good_sample(period * 1000); /* needs period in ps */
192          sampletr -> sample_width = pec_compute_sample_width(period);
193          if(pec_timed_play(period, patterns, actual_time) != SUCCESS)
194          {
195              returncode = FAILURE;
196              if(lm_error("Play timing failed during sweep test. Clock period = %d.\n",
197                  period) != SUCCESS)
198                  return(FAILURE);
199          }
200      }
201      (void)lm_message("done.\n");
202      return(returncode);
203  }
204
205  /*
206  * int good_parity_check(address)
207  * INPUT: address = word address of pattern memory
208  * OUTPUT: return code = SUCCESS or FAILURE
209  * DESCRIPTION: Checks parity circuitry on PAC and PAM
210  * by writing and reading the location specified by 'address'.
211  * The location is written and read 32 times with a pattern of
212  * shifting ones and zeros.
213  */
214  int
215  good_parity_check(address)
216  register u_long address;
217  {
218      u_long i;
219      int j;
220      u_short temp;
221      int returncode = SUCCESS;
222
223      for(i = 0; i < 32; i++)
224      {
225          Write_word(address, i);
226          temp = Read_word(address);
227          if((pacptr(current_lane) -> high_word_parity_error == TRUE) ||
228              (pacptr(current_lane) -> low_word_parity_error == TRUE))
229          {
230              returncode = FAILURE;
231              Clr_pac_errors(current_lane);
232              if(lm_error("PAC's good parity test. Address= %08x, DATA= %04x.\n",
233                  address, i) != SUCCESS)
234                  return(FAILURE);
235          }
236      }
237  #ifdef list
238      if (temp); /* shnt list up */
239  #endif list
240      return(returncode);

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_diag.c

DATE 5/23/89 PAGE #
TIME 4:41:19 pm 3/46

```

LINE # SOURCE TEXT
241 )
242
243 /* int bad_parity_check(address)
244 *
245 * INPUT: address = word address of pattern memory
246 * OUTPUT: return code = SUCCESS or FAILURE
247 * DESCRIPTION: Checks parity circuitry on PAC and PAM
248 * by writing and reading the location specified by 'address'.
249 * The location is written and read 32 times with a pattern of
250 * shifting ones and zeros. The location is written with the
251 * parity set to 'odd' and then read with the parity set to
252 * 'even'. This should create a parity error, latching the
253 * address. The latched address is compared with the actual
254 * address to see if this circuitry is working.
255 */
256
257 int
258 bad_parity_check(address)
259 register u_long address;
260 {
261     u_long i;
262     int j;
263     u_long read_addr;
264     u_long expect_addr;
265     u_short temp;
266     int returncode = SUCCESS;
267
268     switch(address & 0x02) /* High or low word? */
269     {
270     case 0: /* High word test */
271         for(i=0, j=0; j < 32; i += (1 << (j++ & 0x0f)))
272         {
273             pacptr[current_lane] -> high_word_parity = SET_ODD_PARITY;
274             Write_word(address, i);
275             pacptr[current_lane] -> high_word_parity = SET_EVEN_PARITY;
276             temp = Read_word(address);
277             if((pacptr[current_lane] -> high_word_parity_error == TRUE) &&
278                 (pacptr[current_lane] -> low_word_parity_error == FALSE))
279             {
280                 if((read_addr = pacptr[current_lane] -> parity_error_address) !=
281                     (expect_addr = (address >> 2) & 0x3FFFFFF))
282                 {
283                     returncode = FAILURE;
284                     if(!m_error("PAC parity address test. Expected = %07x, \
285                         Actual = %07x.\n", expect_addr, read_addr) != SUCCESS)
286                         return(FAILURE);
287                 }
288             }
289             else
290             {
291                 returncode = FAILURE;
292                 if(!m_error("PAC 'bad' parity test. Address = %08x, Data = %04x.\n",
293                     address, i) != SUCCESS)
294                     return(FAILURE);
295             }
296             Clear_pac_errors(current_lane);
297         }
298         break;
299     case 2: /* Low word test */
300         for(i=0, j=0; j < 32; i += (1 << (j++ & 0x0f)))
301         {
302             pacptr[current_lane] -> low_word_parity = SET_ODD_PARITY;
303             Write_word(address, i);
304             pacptr[current_lane] -> low_word_parity = SET_EVEN_PARITY;
305             temp = Read_word(address);
306             if((pacptr[current_lane] -> high_word_parity_error == FALSE) &&
307                 (pacptr[current_lane] -> low_word_parity_error == TRUE))
308             {
309                 if((read_addr = pacptr[current_lane] -> parity_error_address) !=
310                     (expect_addr = (address >> 2) & 0x3FFFFFF))
311                 {
312                     returncode = FAILURE;
313                     if(!m_error("PAC parity address test. Expected = %07x, \
314                         Actual = %07x.\n", expect_addr, read_addr) != SUCCESS)
315                         return(FAILURE);
316                 }
317             }
318             else
319             {
320                 returncode = FAILURE;
321                 if(!m_error("PAC 'bad' parity test. Address = %08x, Data = %04x.\n",
322                     address, i) != SUCCESS)
323                     return(FAILURE);
324             }
325             Clear_pac_errors(current_lane);
326         }
327         break;
328     default: /* Bad address */
329         return(FAILURE);
330     }
331
332     if(!m_error("PAC parity test complete.\n"))
333         return(returncode);
334 }
335
336 /* int pam_idprom_test(lane_no)
337 *
338 * INPUT: lane_no = lane number to check PAM ID PROMS in
339 * OUTPUT: return code = SUCCESS or FAILURE
340 * DESCRIPTION: Performs checksum tests on PAM ID PROMs.
341 * The function returns SUCCESS if all of the checksums
342 * are correct, and returns FAILURE if any of the checksum
343 * tests fail.
344 */
345
346 int
347 pam_idprom_test(lane_no)
348 int lane_no;
349 {
350     int temp;
351     int pamno;
352     u_long base_address;
353     int returncode = SUCCESS;
354
355     base_address = pac[lane_no].lane_offset + PAM_ID;
356     for(pamno = 0; pamno < pac[lane_no].num_pams; ++pamno)
357     {
358         if((temp = id_checksum((u_char *) (base_address + pamno * PAM_ID_SPACE + 3)))
359             != ID_CHECKSUM_GOOD)
360             returncode = FAILURE;
361     }
362     return(returncode);
363 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_diag.c

DATE 5/23/89
TIME 4:41:19 pm

PAGE #
4/47

```

LINE # SOURCE TEXT
361 {
362     returncode = FAILURE;
363     if(!m_error("PAM ed ID FROM checksum. Expected = %02x. Actual = %02x.\n",
364         pamao, ID_CHECKSUM_GOOD, temp) != SUCCESS)
365         return(FAILURE);
366 }
367 }
368 return(returncode);
369 }
370
371 /*
372  * int pac_comp_pel_ctrl(patterns_no, bit_no)
373  *
374  * INPUT: patterns_no = patterns number, 0's counting
375  *        bit_no = PEL control bit number (0, 1, or 2)
376  * OUTPUT: return code = SUCCESS or FAILURE
377  * DESCRIPTION: Complements the PEL control bit specified by
378  *        bit_no in the pattern specified by patterns_no. The function
379  *        checks if the pattern number is within pattern memory.
380  */
381 int
382 pac_comp_pel_ctrl(patterns_no, bit_no)
383     u_long patterns_no;
384     int bit_no;
385 {
386     u_long address;
387
388     if((patterns_no < 0) || (patterns_no > (pac(current_lane).num_patterns - 1)))
389     {
390         (void)m_error("Pattern outside of pattern memory.\n");
391         return(FAILURE);
392     }
393     address = pac(current_lane).lane_offset + BANK_2 + (4 * patterns_no);
394     *(u_long *)address = 1 << (4 + bit_no);
395     return(SUCCESS);
396 }
397
398 /*
399  * int pac_fill_pel_ctrl(patterns)
400  *
401  * INPUT: patterns = number of patterns to fill
402  * OUTPUT: return code = SUCCESS or FAILURE
403  * DESCRIPTION: Fills the PEL control bits with a counting
404  *        pattern for the number of patterns specified. Patterns
405  *        control is also written.
406  */
407 int
408 pac_fill_pel_ctrl(patterns)
409     int patterns;
410 {
411     register u_long *sumptr;
412     register int pat_no;
413
414     if((patterns < 4) || (patterns > pac(current_lane).num_patterns))
415     {
416         (void)m_error("Number of patterns invalid.\n");
417         return(FAILURE);
418     }
419     sumptr = (u_long *) (pac(current_lane).lane_offset + BANK_2);
420     for(pat_no = 0; pat_no < patterns; pat_no++)
421     {
422         *sumptr++ = (pat_no & 0x7) << 4;
423     }
424     if(build_patterns_control(0, patterns, STOP_MODE) != SUCCESS)
425     {
426         (void)m_error("Could not build pattern control.\n");
427         return(FAILURE);
428     }
429     return(SUCCESS);
430 }
431
432 /*
433  * int pac_predict_offsets(pattern, error_latency, branch_offset,
434  *        block_offset)
435  *
436  * INPUT: pattern = pattern number with parity error
437  *        error_latency = number of patterns played after parity error
438  *        branch_offset = address of predicted branch offset
439  *        block_offset = address of predicted block offset
440  * OUTPUT: return code = SUCCESS or FAILURE
441  * DESCRIPTION: Gives the specified pattern number, this function
442  *        determines what the contents of the branch address and block offset
443  *        registers would be after a pattern play. The function returns
444  *        FAILURE if the pattern would not be played because of a STOP
445  *        instruction in the block. Note: there must not be a parity error
446  *        in pattern memory when this function is called. Nor should there
447  *        be a parity error condition.
448  */
449 int
450 pac_predict_offsets(pattern, error_latency, branch_offset, block_offset)
451     int pattern;
452     int error_latency;
453     int branch_offset;
454     int block_offset;
455 {
456     u_long *sumptr;
457     int block[2];
458     register int i;
459     int branch_no;
460     int stop_no;
461     int played;
462     int patterns_offset;
463
464     /* Figure out which block number pattern is in */
465     block[0] = pattern / BLOCK_SIZE;
466     /* Figure out offset into block */
467     patterns_offset = pattern % BLOCK_SIZE;
468
469     /* Assign "most probable" values to returned arguments (may be modified
470     later in the function) */
471     *branch_offset = block[0];
472     *block_offset = (patterns_offset + error_latency) % BLOCK_SIZE;
473
474     /* Search for branch and stop instructions */
475     /* (assumes one and only one branch and/or stop per block) */
476     sumptr = (u_long *) (pac(current_lane).lane_offset + BANK_2 +
477         block[0] * BLOCK_SIZE * 4);
478     branch_no = -1;
479     stop_no = -1;
480     for(i = 0; i < BLOCK_SIZE; i++)

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_diag.c

DATE 5/23/89 PAGE #
TIME 4:41:19 pm 5/48

```

LINE # SOURCE TEXT
481 {
482     if((*sampleptr & BRANCH_ALWAYS) != 0)
483     {
484         branch_no = 1;
485         break;
486     }
487     if((*sampleptr++ & STOP) != 0)
488     {
489         stop_no = 1;
490         break;
491     }
492 }
493 if(branch_no == -1) /* no branch found => only a stop command */
494 {
495     /* See if pattern gets played */
496     if((stop_no + STOP_LATENCY) > pattern_offset) /* pattern is played */
497         return(SUCCESS);
498     else /* pattern is not played */
499         return(FAILURE);
500 }
501 /* At this point there is at least a branch command */
502 /* See if pattern with error is played before/during branch */
503 if((branch_no + BRANCH_LATENCY) > pattern_offset) /* pattern is played */
504 {
505     /* See if branch command is reached before halt */
506     if((pattern_offset + error_latency - 2) > branch_no) /* reached */
507     {
508         /* At this point, we can compute the block offset. We still do
509         not know the correct branch offset. The block offset is
510         either correct, or if it occurs near the end of a branch, must
511         be modified. */
512         if((played = pattern_offset - (branch_no + BRANCH_LATENCY -
513             error_latency - 4)) >= 0) /* need to modify block_offset */
514             *block_offset = (played + 0xc) & BLOCK_SIZE;
515         /* Now we must determine the correct block number */
516         /* Look up next block */
517         block[1] = Read_long(pac[current_lane].lane_offset + LINK_OFFSET +
518             block[0] * 4) & 0x7fff;
519         /* See if block 2 is entered before halt */
520         /* First, see how many patterns are played in block 1, if any */
521         if((played = (pattern_offset + error_latency) -
522             (branch_no + BRANCH_LATENCY + 2)) > 0) /* patterns played in block 1 */
523         {
524             /* See if there is a branch command within the first "played"
525             patterns in block 1, to see if we branch into block 2 */
526             sampleptr = (u_long *) (pac[current_lane].lane_offset + BANK_1 +
527                 block[1] * BLOCK_SIZE * 4);
528             for(i = 0; i < played; i++)
529             {
530                 if((*sampleptr++ & BRANCH_ALWAYS) != 0)
531                 {
532                     break;
533                 }
534                 if(i == played) /* Did not get to block 2 */
535                 {
536                     *branch_offset = block[1];
537                     return(SUCCESS);
538                 }
539                 else /* Got to block 2 */
540                 {
541                     *branch_offset = Read_long(pac[current_lane].lane_offset +
542                         LINK_OFFSET + block[1] * 4) & 0x7fff;
543                     return(SUCCESS);
544                 }
545             }
546             else /* no patterns played in block 1 */
547             {
548                 *branch_offset = block[1];
549                 return(SUCCESS);
550             }
551             else /* branch is not reached */
552             {
553                 return(SUCCESS);
554             }
555             else /* pattern is not played */
556             {
557                 return(FAILURE);
558             }
559 }
560 void pac_build_walking_branch()
561 {
562     INPUT: none
563     OUTPUT: none
564     DESCRIPTION: Places branch/always instructions in
565     decreasing locations in each block, for a total number
566     of blocks equal to the number of possible branch locations
567     within a block. The link table is set up appropriately.
568 }
569 void
570 pac_build_walking_branch()
571 {
572     u_long *sampleptr;
573     u_long *linkptr;
574     u_long block;
575     sampleptr = (u_long *) (pac[current_lane].lane_offset + BANK_1 +
576         (4 * (2 * BLOCK_SIZE - BRANCH_LATENCY)));
577     linkptr = (u_long *) (pac[current_lane].lane_offset + LINK_OFFSET * 4);
578     /* Put branch instructions in increasing locations in each block */
579     /* and fill up link table */
580     for(block = 0; block < MAX_BRANCHES; ++block)
581     {
582         *sampleptr |= BRANCH_ALWAYS;
583         *sampleptr += BLOCK_SIZE * 2;
584         *linkptr++ = block * 2;
585     }
586 }
587 void setup_good_sample(int)
588 {
589     This sets up a safe sample situation for tests that do not
590     care about the placement of sample, only that "trust" samples
591     do not occur.
592 }
593 void
594 setup_good_sample(period)
595 int period;

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pac_diag.c

DATE 5/23/89

PAGE #

TIME 4:41:19 pm

6/49

```
LINE # SOURCE TEXT
601 {
602     int ramp;
603
604     /* find the fastest safe ramp to use for Edge */
605     for(ramp = 0; ramp < 3; ramp++)
606         if(calib.EdgeMaxDelay[ramp] > period)
607             break;
608
609     /* set up some stuff on the TMC */
610     tmcptr->edge_delay_range = ramp;
611     tmcptr->sample_delay_range = 0; /* choose fastest ramp */
612     tmcptr->sample_mode = EDGE7SAMPLETRIGGERMODE;
613     tmcptr->sample_trigger_threshold = calib.Edge7MinThresh[ramp];
614     tmcptr->sample_delay = calib.SampleMinThresh(0);
615 }
616 }
```

Copyright 1989
Logic Modeling Systems

HEADER FILE
diags/pac_extn.h

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:19 pm | 1/50 |

| LINE # | HEADER TEXT |
|--------|--|
| 1 | /* SCCS_ID: pac_extn.h rev 3.1, 4/24/89 at 07:49:07 */ |
| 2 | |
| 3 | /* |
| 4 | pac_extn.h |
| 5 | |
| 6 | External declarations of global variables |
| 7 | used in PAC diagnostics |
| 8 | |
| 9 | |
| 10 | |
| 11 | /* External variables */ |
| 12 | extern PAC *pacptr(NUMBER_OF_LANES); /* PAC board (one for each lane) */ |
| 13 | extern PAC_INFO pac(NUMBER_OF_LANES); /* PAC information table */ |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pac_glbl.c

DATE

5/23/89

PAGE #

TIME

4:41:19 pm

1/51

| LINE # | SOURCE TEXT |
|--------|--|
| 1 | /* SCSS_ID: pac_glbl.c rev 3.1, 4/24/89 at 07:49:10 */ |
| 2 | /* |
| 3 | |
| 4 | pac_glbl.c |
| 5 | |
| 6 | Global variables |
| 7 | used in PAC diagnosis |
| 8 | |
| 9 | |
| 10 | #include "common.h" |
| 11 | #include "mod_def.h" |
| 12 | #include "pac.h" |
| 13 | #include "pac_def.h" |
| 14 | |
| 15 | PAC "pacptr(NUMBER_OF_LANES), /* PAC boards (one for each lane) */ |
| 16 | PAC_INFO pac(NUMBER_OF_LANES), /* PAC information table */ |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_memtest.c

DATE 5/23/89
TIME 4:41:19 pm

PAGE #
1/52

```

1  /* SCCS ID: pac_memtest.c rev 3.1, 4/24/89 at 07:49:15 */
2  .....
3  *   pac_memtest.c
4  *   .....
5  *   Memory test routines
6  *   used in PAC diagnostics
7  *   .....
8  *   .....
9  *   .....
10 #include "common.h"
11 #include "mod_def.h"
12 #include "modeler_exta.h"
13 #include "pac.h"
14 #include "pac_def.h"
15 #include "pac_exta.h"
16
17
18 typedef struct
19 {
20     u_long bad_data_bits;
21     u_long vram1thru24;
22     u_long vram27thru50;
23     int bad_parity_errors;
24 } MEM_ERRORS;
25
26
27
28 /* void pac_init_mem_err(error_ptr)
29 *
30 * INPUT: error_ptr = pointer to memory error structure
31 * OUTPUT: none
32 * DESCRIPTION: Initializes pettara memory test error structure.
33 */
34 void
35 pac_init_mem_err(error_ptr)
36 MEM_ERRORS *error_ptr;
37 {
38     error_ptr->bad_data_bits = 0L;
39     error_ptr->vram1thru24 = 0L;
40     error_ptr->vram27thru50 = 0L;
41     error_ptr->bad_parity_errors = FALSE;
42 }
43
44
45
46 /* void pac_display_mem_err(error_ptr)
47 *
48 * INPUT: error_ptr = pointer to memory error structure
49 * OUTPUT: none
50 * DESCRIPTION: Displays pettara memory test error structure.
51 */
52 void
53 pac_display_mem_err(error_ptr)
54 MEM_ERRORS *error_ptr;
55 {
56     register int i;
57     register u_long bad_ics;
58     register int word;
59     register int sum_printed = 0;
60
61     (void)lm_message("\nPettara Memory Failure Summary:\n\n");
62     if(error_ptr->bad_data_bits != 0)
63     {
64         (void)lm_message("Bits which did not compare: %08X\n",
65             error_ptr->bad_data_bits);
66     }
67     for(word = 0; word < 2; ++word) /* word=0 => low word */
68     {
69         bad_ics = (word) ? error_ptr->vram1thru24 : error_ptr->vram27thru50;
70         if(bad_ics != 0)
71         {
72             (void)lm_message("\n%08X word memory ICs:\n", (word) ? "High" : "Low");
73             sum_printed = 0;
74             for(i = 1; i < 25; ++i)
75             {
76                 if(((1 << i) & bad_ics) != 0)
77                 {
78                     (void)lm_message("  %04d", (word) ? i : i + 26);
79                     ++sum_printed;
80                 }
81                 if(sum_printed > 12)
82                 {
83                     (void)lm_message("\n");
84                     sum_printed = 0;
85                 }
86             }
87         }
88     }
89     if(sum_printed != 0)
90     {
91         (void)lm_message("\n");
92     }
93     if(error_ptr->bad_parity_errors == TRUE)
94     {
95         (void)lm_message("Parity memory is suspect.\n");
96     }
97 }
98
99 /* int data_bus_test(address, error_ptr)
100 *
101 * INPUT: address = memory address
102 * error_ptr = pointer to memory error structure
103 * OUTPUT: returns SUCCESS or FAILURE
104 * DESCRIPTION: Performs data bus test on a 32-bit memory location.
105 */
106 int
107 data_bus_test(address, error_ptr)
108 u_long address;
109 MEM_ERRORS *error_ptr;
110 {
111     register u_long *memptr = (u_long *)address;
112     register int i;
113     register u_long expected;
114     register u_long actual;
115     register int returncode = SUCCESS;
116     int invert_data;
117
118     for(invert_data = 0; invert_data < 2; invert_data++)
119     {
120         for(i = 1; i <= (1 << 31); i <= 1)
121         {
122             expected = (invert_data) ? ~i : i;
123             *memptr = expected;
124         }
125     }
126 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_memtest.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:19 pm | 2/53 |

```

121  if((actual - *memptr) != expected)
122  {
123      returncode = FAILURE;
124      if(pac_mem_error((u_long)(memptr - 1), expected, actual, error_ptr)
125         != SUCCESS)
126          returncode = FAILURE;
127  }
128  }
129  }
130  return(returncode);
131  }
132  }
133  }
134  /*
135  *   int vram_test(pam_no, error_ptr)
136  *   INPUT:  pam_no = PAM number
137  *           error_ptr = pointer to memory error structure
138  *   OUTPUT: returns SUCCESS or FAILURE
139  *   DESCRIPTION: Performs data bus test all video ram on specified PAM,
140  *               excluding parity ram.
141  */
142  int
143  vram_test(pam_no, error_ptr)
144  int pam_no;
145  MEM_ERRORS *error_ptr;
146  {
147      register int bank_no;
148      register u_long pam_base_address;
149      register u_long bank_base_address;
150      register int i;
151      int returncode = SUCCESS;
152
153      pam_base_address = Pam_base_address(current_lane, pam_no);
154      for(bank_no = 0; bank_no < 3; bank_no++)
155      {
156          bank_base_address = (u_long)get_base_address(pam_no, bank_no);
157          /* test the "even" and "odd" ICs in the bank */
158          for(i = 0; i < 4; i += 4)
159          {
160              if(data_bus_test(bank_base_address + i, error_ptr) != SUCCESS)
161              {
162                  returncode = FAILURE;
163                  if(!error_ptr) data_bus_test_fails |= (1 << i);
164                  return(FAILURE);
165              }
166          }
167      }
168      return(returncode);
169  }
170  }
171  }
172  /*
173  *   int vram_parity_test(pam_no, error_ptr)
174  *   INPUT:  pam_no = PAM number
175  *           error_ptr = pointer to memory error structure
176  *   OUTPUT: returns SUCCESS or FAILURE
177  *   DESCRIPTION: Performs data bus test on all parity video rams
178  *               on selected PAM
179  */
180  int
181  vram_parity_test(pam_no, error_ptr)
182  int pam_no;
183  MEM_ERRORS *error_ptr;
184  {
185      register int bank_no;
186      register int bit_pos;
187      register u_long pam_base_address;
188      register u_long ram_address;
189      register int i;
190      int returncode = SUCCESS;
191
192      Clear_pac_errors(current_lane);
193      pam_base_address = Pam_base_address(current_lane, pam_no);
194      /* test the four parity ICs on the board */
195      for(i = 0; i < 4; i += 2)
196      {
197          ram_address = pam_base_address + i;
198          /* Write and read walking bit in parity ram (bank selects bit i) */
199          for(bit_pos = 0; bit_pos < 3; bit_pos++)
200          {
201              /* write the bits */
202              for(bank_no = 0; bank_no < 3; bank_no++)
203              {
204                  Write_word(ram_address + (bank_no * PAT_MEM_BANK) +
205                           (bank_no * bit_pos));
206              }
207              /* Read the bits and check for parity errors */
208              for(bank_no = 0; bank_no < 3; bank_no++)
209              {
210                  (void)Read_word(ram_address + (bank_no * PAT_MEM_BANK));
211                  if(pac_parity_check(error_ptr) != SUCCESS)
212                      returncode = FAILURE;
213              }
214          }
215      }
216      return(returncode);
217  }
218  }
219  }
220  /*
221  *   int pac_test_a_pam(pam_no)
222  *   INPUT:  pam_no = pattern memory to test (0,1,2,or3)
223  *   OUTPUT: returns SUCCESS or FAILURE
224  *   DESCRIPTION: Performs pattern memory test on specified PAM.
225  *               The procedure is as follows:
226  *   1) Perform an address line test
227  *   2) Initialize the error structure and set the high and low
228  *       parity generators to produce even parity.
229  *   3) Clear pattern memory
230  *   4) Step through pattern memory, one bank at a time,
231  *       reading zeros and writing ones. This test checks
232  *       addressing and ability to clear all memory bits
233  *       (stored as 1's in VRAM's).
234  *   5) Step through pattern memory, one bank at a time,
235  *       reading ones and checking for parity errors after
236  *       each bank. This test checks ability to set all
237  *       memory bits (stored as 0's in VRAM's), and parity
238  *       VRAMS.
239  *   6) Now set the high and low parity generators to produce
240  */

```

| | | | | |
|--|--|---------------------------------------|--------------------|----------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pac_memtest.c | DATE 5/23/89 | PAGE # 3/54 |
| | | | TIME 4:41:19 pm | |
| LINE # | SOURCE TEXT | | | |
| 241 | /* odd parity (default). Then clear all of pattern memory. | | | |
| 242 | /* This puts memory back in same state that it was in after | | | |
| 243 | /* step 3, except the parity memory bits are inverted. | | | |
| 244 | /* 5) Step through pattern memory, one bank at a time, | | | |
| 245 | /* reading zeros and checking for parity errors after | | | |
| 246 | /* each bank. This test checks opposite state of parity VRAMS. | | | |
| 247 | /* 7) Now perform a data bus test on each VRAM. This entails | | | |
| 248 | /* checking each bank, even and odd (6 locations), testing | | | |
| 249 | /* eight ICs at each (4 bits each) by performing a walking | | | |
| 250 | /* ones test followed by a walking zeros test. | | | |
| 251 | /* 8) Finally, perform a walking ones/walking zeros test in | | | |
| 252 | /* each parity VRAM. | | | |
| 253 | /* | | | |
| 254 | int | | | |
| 255 | pac_test_a_pam(pam_no) | | | |
| 256 | int pam_no; | | | |
| 257 | { | | | |
| 258 | MEM_ERRORS mem_errors; *err_ptr = &mem_errors; | | | |
| 259 | int returncode = SUCCESS; | | | |
| 260 | | | | |
| 261 | /* Perform an address line test */ | | | |
| 262 | if(pam_addr_test(pam_no) != SUCCESS) | | | |
| 263 | { | | | |
| 264 | returncode = FAILURE; | | | |
| 265 | if(!error("Address line test on PAM %d failed.\n", pam_no) != SUCCESS) | | | |
| 266 | return FAILURE; | | | |
| 267 | } | | | |
| 268 | | | | |
| 269 | /* Initialize the pattern memory error structure */ | | | |
| 270 | pac_init_mem_err(err_ptr); | | | |
| 271 | | | | |
| 272 | /* Set PAC for even parity */ | | | |
| 273 | pac_set_parity(current_lase, SET_EVEN_PARITY); | | | |
| 274 | | | | |
| 275 | /* Clear pattern memory */ | | | |
| 276 | (void)pac_clear_a_pam(current_lase, pam_no); | | | |
| 277 | | | | |
| 278 | if(pac_read0s_writes(pam_no, err_ptr) != SUCCESS) | | | |
| 279 | { | | | |
| 280 | returncode = FAILURE; | | | |
| 281 | if(!error("Pattern Memory %d fails \"read 0's, write 1's\" test.\n", | | | |
| 282 | pam_no) != SUCCESS) | | | |
| 283 | goto cleanup; | | | |
| 284 | } | | | |
| 285 | if(pac_read_value(pam_no, "01, err_ptr) != SUCCESS) | | | |
| 286 | { | | | |
| 287 | returncode = FAILURE; | | | |
| 288 | if(!error("Pattern Memory %d fails \"read 1's\" test.\n", pam_no) | | | |
| 289 | != SUCCESS) | | | |
| 290 | goto cleanup; | | | |
| 291 | } | | | |
| 292 | /* Set PAC back to odd parity */ | | | |
| 293 | pac_set_parity(current_lase, SET_ODD_PARITY); | | | |
| 294 | | | | |
| 295 | /* Clear pattern memory */ | | | |
| 296 | (void)pac_clear_a_pam(current_lase, pam_no); | | | |
| 297 | | | | |
| 298 | if(pac_read_value(pam_no, "01, err_ptr) != SUCCESS) | | | |
| 299 | { | | | |
| 300 | returncode = FAILURE; | | | |
| 301 | if(!error("Pattern Memory %d fails \"read 0's\" test.\n", pam_no) | | | |
| 302 | != SUCCESS) | | | |
| 303 | goto cleanup; | | | |
| 304 | } | | | |
| 305 | if(vram_test(pam_no, err_ptr) != SUCCESS) | | | |
| 306 | { | | | |
| 307 | returncode = FAILURE; | | | |
| 308 | if(!error("Pattern Memory %d data bus test fails.\n", pam_no) | | | |
| 309 | != SUCCESS) | | | |
| 310 | goto cleanup; | | | |
| 311 | } | | | |
| 312 | /* See if parity errors occurred during the previous tests */ | | | |
| 313 | if(err_ptr->had_parity_errors == TRUE) | | | |
| 314 | { | | | |
| 315 | returncode = FAILURE; | | | |
| 316 | (void)error("Parity errors occurred during memory tests.\n"); | | | |
| 317 | } | | | |
| 318 | else if(returncode != FAILURE) | | | |
| 319 | /* Test the parity bus: has all previous tests have passed */ | | | |
| 320 | { | | | |
| 321 | if(vram_parity_test(pam_no, err_ptr) != SUCCESS) | | | |
| 322 | { | | | |
| 323 | returncode = FAILURE; | | | |
| 324 | (void)error("Pattern Memory %d parity bus test fails.\n", pam_no); | | | |
| 325 | } | | | |
| 326 | } | | | |
| 327 | cleanup: | | | |
| 328 | | | | |
| 329 | if(returncode != SUCCESS) | | | |
| 330 | { | | | |
| 331 | | | | |
| 332 | /* Restore odd parity and clear all PAC errors */ | | | |
| 333 | pac_set_parity(current_lase, SET_ODD_PARITY); | | | |
| 334 | Clear_pac_errors(current_lase); | | | |
| 335 | pac_display_mem_err(err_ptr); | | | |
| 336 | } | | | |
| 337 | return(returncode); | | | |
| 338 | } | | | |
| 339 | | | | |
| 340 | | | | |
| 341 | /* Address line test */ | | | |
| 342 | pam_addr_test(pam_no) | | | |
| 343 | int pam_no; | | | |
| 344 | { | | | |
| 345 | u_long *base; | | | |
| 346 | register int bank; | | | |
| 347 | register int bit; | | | |
| 348 | register int index; | | | |
| 349 | register int max_bits = bitcount(pac[current_lase].pam_size(pam_no)); | | | |
| 350 | register int high_offset = pac[current_lase].pam_size(pam_no) - 1; | | | |
| 351 | int returncode = SUCCESS; | | | |
| 352 | | | | |
| 353 | /* Clear walking 1's and walking 0's address locations in each bank */ | | | |
| 354 | for (bank = 0; bank < 3; ++bank) | | | |
| 355 | { | | | |
| 356 | base = (u_long *)get_base_address(pam_no, bank); | | | |
| 357 | for (bit = 0; bit < max_bits; ++bit) | | | |
| 358 | { | | | |
| 359 | index = 1 << bit; | | | |
| 360 | base[index] = 01; | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_memtest.c

DATE 5/23/89
TIME 4:41:19 pm

PAGE #
4/55

```

LINE #          SOURCE TEXT
361      base[high_offset + index] = 01;
362      }
363      }
364
365      for (bank = 0; bank < 3; ++bank)
366      {
367          base = (u_long *)get_base_address(pam_no, bank);
368          /* Walking 1's */
369          /* Verify that values are cleared */
370          if(pac_verify_clear(pam_no, base, max_bits, 01, &returncode) != SUCCESS)
371              return(Failure);
372          base[0] = 01;
373          if (base[0] != 01) /* verify that base address was set */
374          {
375              returncode = FAILURE;
376              if(lm_error("PAM %d: Address %08X: Expected 00X, read %08X.\n",
377                  pam_no, base, "01, base[0]")) != SUCCESS)
378                  return FAILURE;
379          }
380          /* Verify that values are still cleared */
381          if(pac_verify_clear(pam_no, base, max_bits, 01, &returncode) != SUCCESS)
382              return(Failure);
383          /* Walking 0's */
384          if(pac_verify_clear(pam_no, base, max_bits, high_offset, &returncode)
385              != SUCCESS)
386              return(Failure);
387          base[high_offset] = 01;
388          if (base[high_offset] != 01) /* verify that high address was set */
389          {
390              returncode = FAILURE;
391              if(lm_error("PAM %d: Address %08X: Expected 00X, read %08X.\n",
392                  pam_no, base[high_offset], "01, base[high_offset]")) != SUCCESS)
393                  return FAILURE;
394          }
395          if(pac_verify_clear(pam_no, base, max_bits, high_offset, &returncode)
396              != SUCCESS)
397              return(Failure);
398          }
399      }
400      return(returncode);
401  }
402
403  int
404  pac_verify_clear(pam_no, base, max_bits, offset, returncode)
405  {
406      int pam_no;
407      u_long base;
408      int max_bits;
409      int offset;
410      int *returncode;
411      {
412          register int bit;
413          register int index;
414          for (bit = 0; bit < max_bits; ++bit)
415          {
416              index = offset + (1 << bit);
417              if (base[index] != 01)
418              {
419                  *returncode = FAILURE;
420                  if(lm_error("PAM %d: Address %08X: Expected 0, read %08X.\n",
421                      pam_no, base[index], base[index])) != SUCCESS)
422                      return(Failure);
423              }
424          }
425          return(SUCCESS);
426      }
427  }
428
429  int
430  get_base_address(board, bank)
431  {
432      return Patterns_to_address(current_lase, bank,
433          pac_get_first_pattern_no(current_lase, board));
434  }
435
436  int
437  bitcount(x)
438  {
439      register int count;
440      for (count = 0; x = (x >> 1); ++count)
441          ;
442      return(count);
443  }
444
445  /*
446  int pac_read0s_writals(pam_no, err_ptr)
447  {
448      INPUT:  pam_no = PAM number
449             err_ptr = pointer to memory error structure
450      OUTPUT: returncode SUCCESS or FAILURE
451      DESCRIPTION: Performs test on pattern memory. Reads zeroes
452                  and writes ones. The pattern memory must be zeroed before this
453                  routine is run.
454  */
455  int
456  pac_read0s_writals(pam_no, err_ptr)
457  {
458      int pam_no;
459      MEM_ERRORS *err_ptr;
460      {
461          register u_long *memptr;
462          register u_long temp;
463          register u_long i;
464          register int bank_no;
465          register int returncode = SUCCESS;
466          (void)lm_message("Reading 0's, writing 1's");
467          for(bank_no = 0; bank_no < 3; bank_no++)
468          {
469              lm_message("\n"); /* walking period before each bank */
470              memptr = (u_long *)get_base_address(pam_no, bank_no);
471              i = pac(current_lase).pam_size(pam_no);
472              do
473              {
474                  if((temp = *memptr) != 01)
475                  {
476                      returncode = FAILURE;
477                      if(pac_mem_error((u_long)memptr, 01, temp, err_ptr) != SUCCESS)
478                          return(returncode);
479                  }
480              }

```


| | | | | |
|--|--|---------------------------------------|--------------------|----------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pac_memtest.c | DATE 5/23/89 | PAGE # 5/56 |
| | | | TIME 4:41:19 pm | |

| LINE # | SOURCE TEXT |
|--------|--|
| 481 | *memptr++ = 01; |
| 482 | } while (--i); |
| 483 | } |
| 484 | (void)lm_message("Done.\n"); |
| 485 | return(returncode); |
| 486 | } |
| 487 | |
| 488 | /* |
| 489 | int pac_parity_error_check(err_ptr) |
| 490 | /* |
| 491 | INPUT: err_ptr = pointer to memory error structure |
| 492 | OUTPUT: returns SUCCESS or FAILURE |
| 493 | DESCRIPTION: Determines if a parity error was detected during |
| 494 | a memory test read. If not, simply returns. If an error did |
| 495 | occur, displays address and whether high word, low word, or |
| 496 | both caused error. Fills in error structure and clears error. |
| 497 | */ |
| 498 | int |
| 499 | pac_parity_error_check(err_ptr) |
| 500 | MEM_ERRORS *err_ptr; |
| 501 | { |
| 502 | int returncode = SUCCESS; |
| 503 | if(pacptr[current_lane] == high_word_parity_error == 1) |
| 504 | { |
| 505 | (void)lm_error("High word parity error reading address %08X.\n", |
| 506 | Pac_parity_address(current_lane)); |
| 507 | returncode = FAILURE; |
| 508 | } |
| 509 | if(pacptr[current_lane] == low_word_parity_error == 1) |
| 510 | { |
| 511 | (void)lm_error("Low word parity error reading address %08X.\n", |
| 512 | Pac_parity_address(current_lane)); |
| 513 | returncode = FAILURE; |
| 514 | } |
| 515 | if(returncode != SUCCESS) |
| 516 | { |
| 517 | /* Log into error structure and clear errors */ |
| 518 | err_ptr->had_parity_errors = TRUE; |
| 519 | Clear_pac_errors(current_lane); |
| 520 | } |
| 521 | return(returncode); |
| 522 | } |
| 523 | |
| 524 | /* |
| 525 | int pac_read_value(pam_no, value, err_ptr) |
| 526 | /* |
| 527 | INPUT: pam_no = PAM number |
| 528 | value = 32-bit value to read |
| 529 | err_ptr = pointer to memory error structure |
| 530 | OUTPUT: returns SUCCESS or FAILURE |
| 531 | DESCRIPTION: Performs test on pattern memory. Reads value |
| 532 | specified. The pattern memory must be set to value before this |
| 533 | routine is run. |
| 534 | */ |
| 535 | int |
| 536 | pac_read_value(pam_no, value, err_ptr) |
| 537 | int pam_no; |
| 538 | register u_long value; |
| 539 | MEM_ERRORS *err_ptr; |
| 540 | { |
| 541 | register u_long *memptr; |
| 542 | register u_long temp; |
| 543 | register u_long i; |
| 544 | int returncode = SUCCESS; |
| 545 | int bank_no; |
| 546 | (void)lm_message("Reading %08X's", value); |
| 547 | for(bank_no = 0; bank_no < 3; bank_no++) |
| 548 | { |
| 549 | lm_message(" ", /* walking period before each bank */ |
| 550 | memptr = (u_long *)get_base_address(pam_no, bank_no); |
| 551 | i = pac[current_lane].pam_size[pam_no]; |
| 552 | bank_status = SUCCESS; |
| 553 | do |
| 554 | { |
| 555 | if((temp = *memptr++) != value) |
| 556 | { |
| 557 | bank_status = FAILURE; |
| 558 | if(pac_mem_error((u_long)(memptr - 1), value, temp, err_ptr) != SUCCESS) |
| 559 | return(FAILURE); |
| 560 | } |
| 561 | } while (--i); |
| 562 | if(bank_status == SUCCESS) |
| 563 | { |
| 564 | (void)pac_parity_error_check(err_ptr); |
| 565 | } |
| 566 | else |
| 567 | { |
| 568 | returncode = FAILURE; |
| 569 | } |
| 570 | (void)lm_message("Done.\n"); |
| 571 | return(returncode); |
| 572 | } |
| 573 | /* |
| 574 | int pac_link_memtest() |
| 575 | /* |
| 576 | INPUT: none |
| 577 | OUTPUT: returns SUCCESS or FAILURE |
| 578 | DESCRIPTION: Performs link table memory test. |
| 579 | */ |
| 580 | int |
| 581 | pac_link_memtest() |
| 582 | { |
| 583 | void pac_clear_link(); |
| 584 | register u_long *memptr; |
| 585 | register u_long temp; |
| 586 | register u_long i; |
| 587 | u_long base_address; |
| 588 | int returncode = SUCCESS; |
| 589 | /* First clear link table */ |
| 590 | pac_clear_link(); |
| 591 | base_address = pac[current_lane].lane_offset + LINK_OFFSET; |
| 592 | /* Next perform a read 0's, write 1's test */ |
| 593 | memptr = (u_long *)base_address; |
| 594 | i = LINK_SIZE >> 2; |
| 595 | do |
| 596 | { |
| 597 | if((temp = ((*memptr) & 0xffff)) != 01) |
| 598 | { |
| 599 | returncode = FAILURE; |
| 600 | } |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pac_memtest.c | DATE 5/23/89 | PAGE # 6/57 |
|--|--|---|-----------------|----------------|
| LINE # | | SOURCE TEXT | | |
| 601 | | returncode = FAILURE; | | |
| 602 | | if(!m_error("Link table memory failure at address %08x.\n", | | |
| 603 | | tExpected = %04x, actual = %04x.\n", 0, memptr, temp) != SUCCESS) | | |
| 604 | | return(FAILURE); | | |
| 605 | | { | | |
| 606 | | memptr++ = 01; | | |
| 607 | | while (--i); | | |
| 608 | | } | | |
| 609 | | /* Next perform a read 1's, write 0's test */ | | |
| 610 | | memptr = (u_long *) (base_address); | | |
| 611 | | i = LINK_SIZE >> 2; | | |
| 612 | | do | | |
| 613 | | { | | |
| 614 | | if((temp = (*(memptr & 0xffff)) != 0xffff)) | | |
| 615 | | { | | |
| 616 | | returncode = FAILURE; | | |
| 617 | | if(!m_error("Link table memory failure at address %08x.\n", | | |
| 618 | | tExpected = %04x, actual = %04x.\n", 0xffff, memptr, temp) != SUCCESS) | | |
| 619 | | return(FAILURE); | | |
| 620 | | } | | |
| 621 | | memptr++ = 01; | | |
| 622 | | while (--i); | | |
| 623 | | } | | |
| 624 | | /* Now perform a location test on the first link table location */ | | |
| 625 | | if(location_test(base_address, 0, 16) != SUCCESS) | | |
| 626 | | { | | |
| 627 | | returncode = FAILURE; | | |
| 628 | | (void)m_error("Link table data bus test fails.\n"); | | |
| 629 | | } | | |
| 630 | | return(returncode); | | |
| 631 | | } | | |
| 632 | | } | | |
| 633 | | /* | | |
| 634 | | int pac_mem_error(address, expected, actual, mem_error_ptr) | | |
| 635 | | { | | |
| 636 | | /* INPUT: address = memory address of failed location | | |
| 637 | | expected = expected value of failed location | | |
| 638 | | actual = actual value of failed location | | |
| 639 | | mem_error_ptr = pointer to memory error structure | | |
| 640 | | /* OUTPUT: returns SUCCESS or FAILURE (return of m_error) | | |
| 641 | | /* DESCRIPTION: Outputs error message based on arguments. | | |
| 642 | | Computes U-number of suspect memory IC(s). Fills in | | |
| 643 | | memory error structure; | | |
| 644 | | */ | | |
| 645 | | int | | |
| 646 | | pac_mem_error(address, expected, actual, mem_error_ptr) | | |
| 647 | | { | | |
| 648 | | register u_long address; | | |
| 649 | | register u_long expected; | | |
| 650 | | register u_long actual; | | |
| 651 | | MEM_ERRORS *mem_error_ptr; | | |
| 652 | | { | | |
| 653 | | register u_long ref_des; | | |
| 654 | | register u_long bad_bits; | | |
| 655 | | register int nibble; | | |
| 656 | | u_long base; | | |
| 657 | | char buffer[(4 * 8) + 1]; | | |
| 658 | | char *bufptr = buffer; | | |
| 659 | | bad_bits = expected ^ actual; | | |
| 660 | | mem_error_ptr->bad_data_bits = bad_bits; | | |
| 661 | | { | | |
| 662 | | base = 1 + ((2 - ((address >> 26) & 3)) << 1) + (1 - ((address >> 2) & 1)); | | |
| 663 | | for(nibble = 0; nibble < 8; nibble++) | | |
| 664 | | { | | |
| 665 | | if(((bad_bits >> (nibble << 2)) & 0xf) != 0) | | |
| 666 | | { | | |
| 667 | | ref_des = base + (26 * (1 - nibble / 4) + 6 + (3 - nibble & 4)); | | |
| 668 | | /* Fill in memory error structure with ref_des */ | | |
| 669 | | if(ref_des < 27) | | |
| 670 | | mem_error_ptr->vramalt024 = 1 << ref_des; | | |
| 671 | | else | | |
| 672 | | mem_error_ptr->vramalt024 = 1 << (ref_des - 26); | | |
| 673 | | /* Now fill up buffer for printing */ | | |
| 674 | | bufptr++ = 'U'; | | |
| 675 | | if(ref_des > 9) | | |
| 676 | | { | | |
| 677 | | bufptr++ = '0' + (char)(ref_des / 10); | | |
| 678 | | bufptr++ = '0' + (char)(ref_des % 10); | | |
| 679 | | } | | |
| 680 | | else | | |
| 681 | | bufptr++ = '0' + (char)(ref_des); | | |
| 682 | | bufptr++ = ' '; | | |
| 683 | | } | | |
| 684 | | } | | |
| 685 | | bufptr = '\0'; | | |
| 686 | | return(m_error("Memory error: Addr=%08X Bits=%08X RAMS=%s\n", | | |
| 687 | | address, bad_bits, buffer)); | | |
| 688 | | } | | |
| 689 | | } | | |
| 690 | | } | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_menu.c

DATE 5/23/89
TIME 4:41:20 pm

PAGE #
1/58

```

1  /* SCCS ID: pac_menu.c rev 1.1, 4/24/89 at 07:49:19 */
2  /*****
3  *
4  *   pac_menu.c
5  *
6  *   Main Menu
7  *   used in Pattern Controller diagnostics
8  *
9  * *****/
10 #include "common.h"
11 #include "lm_diags.h"
12 #include "mod_def.h"
13 #include "moduler_extn.h"
14 #include "pac_def.h"
15 #include "pac.h"
16 #include "pac_extn.h"
17
18 pac_diag_disp(parent_menu)
19 LM_DIAG_MENU *parent_menu;
20 {
21     int pac_reg_test();
22     int pac_link_mem_test();
23     int pac_idprom_test();
24     int pac_detect_pams();
25     int pac_check_pamid();
26     int pac_pat_mem_disp();
27     int pac_strapping_test();
28     int pac_cpu_parity_test();
29     int pac_pattern_disp();
30     int pac_util_disp();
31
32     static LM_DIAG_MENU_ITEM menu_list[] =
33     {
34         {
35             "1",
36             "Pattern Controller Register Test",
37             pac_reg_test,
38             LM_DIAG_diag_routine,
39             LM_DIAG_null
40         },
41         {
42             "2",
43             "Link Table Test",
44             pac_link_mem_test,
45             LM_DIAG_diag_routine,
46             LM_DIAG_null
47         },
48         {
49             "3",
50             "Pattern Controller ID Prom Test",
51             pac_idprom_test,
52             LM_DIAG_diag_routine,
53             LM_DIAG_null
54         },
55         {
56             "4",
57             "Pattern Memory Detection Test",
58             pac_detect_pams,
59             LM_DIAG_diag_routine,
60             LM_DIAG_null
61         },
62         {
63             "5",
64             "Pattern Memory ID Prom Test",
65             pac_check_pamid,
66             LM_DIAG_diag_routine,
67             LM_DIAG_null
68         },
69         {
70             "6",
71             "Pattern Memory Strapping Test",
72             pac_strapping_test,
73             LM_DIAG_diag_routine,
74             LM_DIAG_null
75         },
76         {
77             "7",
78             "Pattern Memory Test Menu",
79             pac_pat_mem_disp,
80             LM_DIAG_another_menu,
81             LM_DIAG_null
82         },
83         {
84             "8",
85             "Pattern Controller Parity Circuit Test",
86             pac_cpu_parity_test,
87             LM_DIAG_diag_routine,
88             LM_DIAG_null
89         },
90         {
91             "9",
92             "Pattern Play Test Menu",
93             pac_patterns_disp,
94             LM_DIAG_another_menu,
95             LM_DIAG_null
96         },
97         {
98             "10",
99             "Pattern Controller Utilities Menu",
100            pac_util_disp,
101            LM_DIAG_utility_menu,
102            LM_DIAG_null
103        }
104    };
105
106    static LM_DIAG_MENU pac_main_menu =
107    {
108        "PATTERN CONTROLLER DIAGNOSTICS",
109        sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
110        0,
111        menu_list
112    };
113
114    pac_main_menu.title = parent_menu->
115    menu_items[parent_menu->current_selection].menu_text;
116
117    return lm_display_menu(&pac_main_menu);
118 }
119
120 /* Pattern Memory Test Menu

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_menu.c

DATE 5/23/89
TIME 4:41:20 pm

PAGE #
2/59

```

LINE # SOURCE TEXT
121 //*****
122 pac_pat_mnu_disp(parent_mnu)
123 LM_DIAG_MENU "parent_mnu:
124 {
125     register int pam_no;
126     static char pam_buffer[4][80];
127     int pac_pat_mnu_test();
128     static LM_DIAG_MENU_ITEM mnu_list[] =
129     {
130         {
131             "1",
132             pam_buffer[0],
133             pac_pat_mnu_test,
134             LM_DIAG_diag_routine,
135             LM_DIAG_null,
136             "0"
137         },
138         {
139             "2",
140             pam_buffer[1],
141             pac_pat_mnu_test,
142             LM_DIAG_diag_routine,
143             LM_DIAG_null,
144             "1"
145         },
146         {
147             "3",
148             pam_buffer[2],
149             pac_pat_mnu_test,
150             LM_DIAG_diag_routine,
151             LM_DIAG_null,
152             "2"
153         },
154         {
155             "4",
156             pam_buffer[3],
157             pac_pat_mnu_test,
158             LM_DIAG_diag_routine,
159             LM_DIAG_null,
160             "3"
161         }
162     },
163     static LM_DIAG_MENU mnu =
164     {
165         0,
166         sizeof(mnu_list) / sizeof(LM_DIAG_MENU_ITEM),
167         mnu_list
168     },
169     mnu.title = parent_mnu->
170     mnu.items[parent_mnu->current_selection].mnu_text,
171     for(pam_no = 0; pam_no < 4; pam_no++)
172     {
173         if(pac(current_lane).mnu_pam > pam_no) /* enable test */
174         {
175             mnu_list[pam_no].attributes |= LM_DIAG_disable;
176             sprintf(pam_buffer[pam_no], "PAM to Memory Test (tdk)", pam_no,
177                 pac(current_lane).pam_size[pam_no] >> 10);
178         }
179         else /* disable test */
180         {
181             mnu_list[pam_no].attributes |= LM_DIAG_disable;
182             sprintf(pam_buffer[pam_no], "PAM to Memory Test ( - )", pam_no);
183         }
184     }
185     return lm_display_mnu(mnu);
186 }
187 //*****
188 * Pattern Play Menu
189 //*****
190 #define MENU_INDEX_PAT_BIT 3
191 #define MENU_INDEX_PAT_BUS 4
192 #define MENU_INDEX_CRC 5
193
194 pac_patterns_disp(parent_mnu)
195 LM_DIAG_MENU "parent_mnu:
196 {
197     int pals_found;
198     int pac_freq_test();
199     int pac_error_test();
200     int pac_branch_test();
201     int pac_patterns_bits_test();
202     int pac_patterns_bus_test();
203     int pac_crc_test();
204     static LM_DIAG_MENU_ITEM mnu_list[] =
205     {
206         {
207             "1",
208             "Frequency Sweep Test",
209             pac_freq_test,
210             LM_DIAG_diag_routine,
211             LM_DIAG_null
212         },
213         {
214             "2",
215             "Parity Error Test",
216             pac_error_test,
217             LM_DIAG_diag_routine,
218             LM_DIAG_null
219         },
220         {
221             "3",
222             "Fast Branching Test",
223             pac_branch_test,
224             LM_DIAG_diag_routine,
225             LM_DIAG_null
226         },
227         {
228             "4",
229             "Pattern Bit Test",
230             pac_patterns_bits_test,
231             LM_DIAG_diag_routine,
232             LM_DIAG_diag_routine
233         }
234     }
235 }
236

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_menu.c

DATE 5/23/89
TIME 4:41:20 pm

PAGE #
3/60

LINE # SOURCE TEXT

```

241 LM_DIAG_null
242 },
243 {
244     "g",
245     "Pattern Bus Test",
246     pac_pattern_bus_test,
247     LM_DIAG_diag_routine,
248     LM_DIAG_null
249 },
250 },
251 "g",
252 "Diagnostic Adapter CRC Test",
253 pac_crc_test,
254 LM_DIAG_diag_routine,
255 LM_DIAG_null
256 },
257 },
258
259 static LM_DIAG_MENU pac_play_menu =
260 {
261     "PATTERN PLAY SUB-MENU",
262     sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
263     0,
264     menu_list
265 },
266 pac_play_menu.title = parent_menu->
267 menu_item[parent_menu->current_selection].menu_text,
268
269 if((pala_found = pac_check_for_pala()) == 0) /* No PELA in lane */
270 {
271     menu_list[MENU_INDEX_PAT_BIT].attributes |= LM_DIAG_disable;
272     menu_list[MENU_INDEX_PAT_BUS].attributes |= LM_DIAG_disable;
273     menu_list[MENU_INDEX_CRC].attributes |= LM_DIAG_disable;
274 }
275 else
276 {
277     menu_list[MENU_INDEX_PAT_BIT].attributes |= LM_DIAG_disable;
278     menu_list[MENU_INDEX_PAT_BUS].attributes |= LM_DIAG_disable;
279     /* Check for diagnostic adapters */
280     if(pac_check_for_diag_dabs(pala_found) == 0) /* No diag dabs in lane */
281     menu_list[MENU_INDEX_CRC].attributes |= LM_DIAG_disable;
282     else
283     menu_list[MENU_INDEX_CRC].attributes |= LM_DIAG_disable;
284 }
285
286 return la_display_menu(&pac_play_menu);
287 }
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pac_menu.c | DATE 5/23/89 TIME 4:41:20 pm | PAGE # 4/61 |
|--|---|------------------------------------|---------------------------------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 361 | /* | | | |
| 362 | Memory Fill Menu | | | |
| 363 | */ | | | |
| 364 | pac_fill_memory_util(parent_menu) | | | |
| 365 | LM_DIAG_MENU "parent_menu" | | | |
| 366 | { | | | |
| 367 | int pac_fill_mem_xero(); | | | |
| 368 | int pac_fill_mem_random(); | | | |
| 369 | int pac_fill_mem_counting(); | | | |
| 370 | int pac_fill_mem_walk1(); | | | |
| 371 | int pac_fill_mem_walk0(); | | | |
| 372 | int pac_fill_mem_lam0(); | | | |
| 373 | static LM_DIAG_MENU_ITEM menu_list[] = | | | |
| 374 | { | | | |
| 375 | {"1", | | | |
| 376 | "Clear Pattern Memory", | | | |
| 377 | pac_fill_mem_xero, | | | |
| 378 | LM_DIAG_utility LM_DIAG_automatic_quit, | | | |
| 379 | LM_DIAG_null | | | |
| 380 | }, | | | |
| 381 | {"2", | | | |
| 382 | "Fill with random data", | | | |
| 383 | pac_fill_mem_random, | | | |
| 384 | LM_DIAG_utility LM_DIAG_automatic_quit, | | | |
| 385 | LM_DIAG_null | | | |
| 386 | }, | | | |
| 387 | {"3", | | | |
| 388 | "Fill with Counting Data", | | | |
| 389 | pac_fill_mem_counting, | | | |
| 390 | LM_DIAG_utility LM_DIAG_automatic_quit, | | | |
| 391 | LM_DIAG_null | | | |
| 392 | }, | | | |
| 393 | {"4", | | | |
| 394 | "Fill with Walking Ones", | | | |
| 395 | pac_fill_mem_walk1, | | | |
| 396 | LM_DIAG_utility LM_DIAG_automatic_quit, | | | |
| 397 | LM_DIAG_null | | | |
| 398 | }, | | | |
| 399 | {"5", | | | |
| 400 | "Fill with Walking Zeros", | | | |
| 401 | pac_fill_mem_walk0, | | | |
| 402 | LM_DIAG_utility LM_DIAG_automatic_quit, | | | |
| 403 | LM_DIAG_null | | | |
| 404 | }, | | | |
| 405 | {"6", | | | |
| 406 | "Fill with Alternating Ones and Zeros", | | | |
| 407 | pac_fill_mem_lam0, | | | |
| 408 | LM_DIAG_utility LM_DIAG_automatic_quit, | | | |
| 409 | LM_DIAG_null | | | |
| 410 | }, | | | |
| 411 | static LM_DIAG_MENU fill_memory_menu = | | | |
| 412 | { | | | |
| 413 | "FILL PATTERN MEMORY MENU", | | | |
| 414 | sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM), | | | |
| 415 | 0, | | | |
| 416 | menu_list | | | |
| 417 | }, | | | |
| 418 | fill_memory_menu.title = parent_menu-> | | | |
| 419 | menu_items[parent_menu->current_selection].menu_text, | | | |
| 420 | return lm_display_menu(&fill_memory_menu); | | | |
| 421 | } | | | |
| 422 | /* | | | |
| 423 | Pattern Play Utility Menu | | | |
| 424 | */ | | | |
| 425 | pac_play_util_disp(parent_menu) | | | |
| 426 | LM_DIAG_MENU "parent_menu" | | | |
| 427 | { | | | |
| 428 | int pac_sel_pat_clk(); | | | |
| 429 | int pac_single_play(); | | | |
| 430 | int pac_loop_with_sample(); | | | |
| 431 | int pac_loop_no_sample(); | | | |
| 432 | int pac_loop_no_pams(); | | | |
| 433 | static LM_DIAG_MENU_ITEM menu_list[] = | | | |
| 434 | { | | | |
| 435 | {"1", | | | |
| 436 | "Select Pattern Clock Frequency", | | | |
| 437 | pac_sel_pat_clk, | | | |
| 438 | LM_DIAG_utility, | | | |
| 439 | LM_DIAG_null | | | |
| 440 | }, | | | |
| 441 | {"2", | | | |
| 442 | "Single Presentation (reports time in ms)", | | | |
| 443 | pac_single_play, | | | |
| 444 | LM_DIAG_utility, | | | |
| 445 | LM_DIAG_null | | | |
| 446 | }, | | | |
| 447 | {"3", | | | |
| 448 | "Looping Play with Sample", | | | |
| 449 | pac_loop_with_sample, | | | |
| 450 | LM_DIAG_utility, | | | |
| 451 | LM_DIAG_null | | | |
| 452 | }, | | | |
| 453 | {"4", | | | |
| 454 | "Looping Play, no Sample", | | | |
| 455 | pac_loop_no_sample, | | | |
| 456 | LM_DIAG_utility, | | | |
| 457 | LM_DIAG_null | | | |
| 458 | }, | | | |
| 459 | {"5", | | | |
| 460 | "Looping Play, Dummy Pattern Memory", | | | |
| 461 | } | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pac_menu.c | DATE 5/23/89 | PAGE # 5/62 |
|--|---|------------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 481 | pac_loop_no_pams, | | | |
| 482 | lm_diag_utility, | | | |
| 483 | lm_diag_null | | | |
| 484 | } | | | |
| 485 | , | | | |
| 486 | static lm_diag_menu play_util_menu = | | | |
| 487 | { | | | |
| 488 | "PATTERN PLAY UTILITIES MENU", | | | |
| 489 | sizeof(menu_list) / sizeof(lm_diag_menu_item), | | | |
| 490 | 0, | | | |
| 491 | menu_list | | | |
| 492 | }; | | | |
| 493 | play_util_menu.title = parent_menu-> | | | |
| 494 | menu_items(parent_menu->current_selection).menu_text, | | | |
| 495 | return lm_display_menu(&play_util_menu); | | | |
| 496 | } | | | |
| 497 | | | | |
| 498 | | | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pac_menu_diag.c

DATE

5/23/89

PAGE #

TIME

4:41:20 pm

1/63

```

1  /* SCCS ID: pac_menu_diag.c rev 3.1, 4/24/89 at 07:49:23 */
2
3  .....
4  *   pac_menu_diag.c
5  *
6  *   Main diagnostic routines called directly by PAC menus
7  *   used in PAC diagnostics
8  *
9  .....
10 #include "vrx.h" /* for in_delay and in_time */
11 #include "common.h"
12 #include "in_diags.h"
13 #include "mod_def.h"
14 #include "moduler_exta.h"
15 #include "tag.h"
16 #include "tag_def.h"
17 #include "tag_exta.h"
18 #include "pac.h"
19 #include "magic.h"
20 #include "pel.h"
21 #include "pac_def.h"
22 #include "pac_exta.h"
23 #include "id.h"
24
25 /*
26  *   int pac_error_test()
27  *
28  *   INPUT: none
29  *   OUTPUT: returns code = SUCCESS or FAILURE
30  *   DESCRIPTION:
31  */
32 int
33 pac_error_test()
34 {
35     void pac_build_walking_branch();
36     int pattern;
37     int timeout;
38     u_long *memptr;
39     int branch_address;
40     int block_offset;
41     int gen_err;
42
43     if(pac_stack_pama(current_lane) != SUCCESS)
44     {
45         (void)lm_error("Pattern memory configuration error.\n");
46         return(FAILURE);
47     }
48
49     if(diag_clear_errors() != SUCCESS)
50         return(FAILURE);
51
52     /* Fill pattern memory with random data */
53     if(pac_fill_random(BLOCK_SIZE, (MAX_BRANCHES + 1) * BLOCK_SIZE, NOP_MASK,
54         SEED) != SUCCESS)
55     {
56         (void)lm_error("Could not fill with random data.\n");
57         return(FAILURE);
58     }
59
60     /* Set up pattern memory for "walking" branching */
61     pac_build_walking_branch();
62     if(pac_set_pattern_clock(PAC_MIN_PERIOD) != SUCCESS)
63     {
64         (void)lm_error("Could not set pattern clock to minimum period.\n");
65         return(FAILURE);
66     }
67
68     setup_good_sample(PAC_MIN_PERIOD * 1000); /* needs period in us */
69     /* Prepare for pattern play and compute a conservative timeout */
70     if((timeout = pac_pre_play()) == 0)
71     {
72         (void)lm_error("Pattern play preparation failed.\n");
73         return(FAILURE);
74     }
75
76     /* Put a stop instruction in the last block */
77     /* so that 10 patterns within the last block are played */
78     memptr = (u_long *) (pac(current_lane).lane_offset + BANK_2 +
79         ((MAX_BRANCHES + 1) * BLOCK_SIZE) + 10 * STOP_LATENCY);
80     *memptr = STOP;
81
82     (void)lm_message("Playing patterns");
83
84     for(pattern = BLOCK_SIZE; pattern < ((MAX_BRANCHES * BLOCK_SIZE) + 1);
85         pattern++)
86     {
87         if((pattern % (BLOCK_SIZE * 2)) == 0)
88         {
89             (void)lm_message("-");
90             gen_err = pac_predict_offsets(pattern, 5, &branch_address, &block_offset);
91             if(pac_insert_parity_error(pattern, 0) != SUCCESS)
92             {
93                 (void)lm_error("Could not insert parity error in pattern control.\n");
94                 return(FAILURE);
95             }
96
97             pacptr[current_lane]-->branch_address = 1;
98             if(pac_play(timeout) != SUCCESS)
99             {
100                 (void)lm_error("Pattern play failed.\n");
101                 return(FAILURE);
102             }
103
104             if(gen_err != SUCCESS)
105             {
106                 if(pacptr[current_lane]-->parity_error == TRUE)
107                 {
108                     (void)lm_error("Pattern play generated error.\n");
109                     return(FAILURE);
110                 }
111             }
112             else
113             {
114                 if(pacptr[current_lane]-->parity_error != TRUE)
115                 {
116                     (void)lm_error("Pattern play did not generate error (should have).\n");
117                     return(FAILURE);
118                 }
119                 if(pacptr[current_lane]-->branch_address != branch_address)
120                 {
121                     (void)lm_error("Branch address incorrect.\n\tExpected %d. Actual %d.\n",
122                         branch_address, pacptr[current_lane]-->branch_address);
123                     return(FAILURE);
124                 }
125                 if(pacptr[current_lane]-->block_offset != block_offset)
126                 {
127                     (void)lm_error("Block offset incorrect.\n\tExpected %d. Actual %d.\n",

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_menu_diag.c

DATE 5/23/89
TIME 4:41:20 pm

PAGE #
2/64

```

LINE # SOURCE TEXT
121     block_offset, pacptr(current_lane) -> block_offset);
122     return(FAILURE);
123 }
124 }
125 if(pac_remove_parity_error(pattern, 0) != SUCCESS)
126 {
127     (void)lm_error("Could not remove parity error in pattern control.\n");
128     return(FAILURE);
129 }
130 if(gen_err != SUCCESS)
131 /* Increment pattern count to next block boundary (-1) */
132 pattern = ((pattern / BLOCK_SIZE) + 1) * BLOCK_SIZE - 1;
133 }
134 (void)lm_message("done.\n");
135 return(SUCCESS);
136 }
137 }
138
139 /*
140  * int pac_cpu_parity_test()
141  *
142  * INPUT: none
143  * OUTPUT: return code = SUCCESS or FAILURE
144  * DESCRIPTION: Performs test on PAC parity
145  *               circuitry used for CPU reads and writes.
146  *               Tests are run on first location in pattern
147  *               memory.
148  */
149 int
150 pac_cpu_parity_test()
151 {
152     if(pac_stack_pane(current_lane) != SUCCESS)
153     {
154         (void)lm_error("Pattern memory configuration error.\n");
155         return(FAILURE);
156     }
157     /* Perform pac_parity_test using first location in Bank 0 of pattern mem */
158     return pac_parity_test(pac(current_lane).lane_offset);
159 }
160
161 /*
162  * int pac_refresh_test()
163  *
164  * INPUT: none
165  * OUTPUT: return code = SUCCESS or FAILURE
166  * DESCRIPTION: Tests refresh of pattern memory
167  *               by writing random data, pausing for 1 second,
168  *               and then reading the data back. This is repeated
169  *               for a user specified number of trials. Each trial
170  *               uses a new seed (random) to generate a different
171  *               set of random numbers.
172  */
173 int
174 pac_refresh_test()
175 {
176     u_long random_seed;
177     u_long i;
178     u_long trials;
179     u_long failures;
180
181     if(diag_clear_errors() != SUCCESS)
182         return(FAILURE);
183
184     failures = 0;
185     random_seed = SEED;
186     trials = 1;
187     diag_get_ulong(&trials, "number of trials", 11, 65535);
188     (void)lm_message("Hit key to exit after a trial.\n");
189     for(i = 0; (i < trials) && (lm_check_key() == 0); ++i)
190     {
191         /* Fill all of pattern memory with random data, using random seed */
192         if(pac_fill_random(0, pac(current_lane).num_patterns, 01,
193             (random_seed = pac_get_random(random_seed))) != SUCCESS)
194         {
195             (void)lm_error("Could not perform refresh test.\n");
196             return(FAILURE);
197         }
198         lm_delay(1000); /* Pause for 1 second */
199         if(pac_read_random(0, pac(current_lane).num_patterns, "01, random_seed")
200             != SUCCESS)
201         {
202             if(lm_error("Refresh test fails random read number %d.\n", i + 1)
203                 != SUCCESS)
204                 return(FAILURE);
205             else
206                 ++failures;
207         }
208         else
209             (void)lm_message("Trial number %d passes.\n", i + 1);
210     }
211     clear_key_buf();
212     if(failures != 0)
213     {
214         (void)lm_error("Refresh test failed %d times.\n", failures);
215         return(FAILURE);
216     }
217     else
218     {
219         (void)lm_message("Refresh test passes.\n");
220         return(SUCCESS);
221     }
222 }
223
224 /*
225  * int pac_branch_test()
226  *
227  * INPUT: none
228  * OUTPUT: return code = SUCCESS or FAILURE
229  * DESCRIPTION:
230  */
231 int
232 pac_branch_test()
233 {
234     void pac_build_fast_branch();
235     int patterns;
236     int returncode = SUCCESS;
237
238     if(pac_stack_pane(current_lane) != SUCCESS)
239     {
240         (void)lm_error("Pattern memory configuration error.\n");

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_menu_diag.c

DATE 5/23/89 PAGE #
TIME 4:41:20 pm 3/65

```

LINE #          SOURCE TEXT
241      }
242      }
243      }
244      if(diag_clear_errors() != SUCCESS)
245      {
246      }
247      patterns = pac(current_lane).num_blocks * (BRANCH_LATENCY + 1);
248      /* Fill all of pattern memory with random data */
249      if(pac_fill_random(0, pac(current_lane).num_patterns, NOP_MASK, SEED)
250      != SUCCESS)
251      {
252      (void)lm_error("Could not perform branch test.\n");
253      return(FAILURE);
254      }
255      /* Set up pattern memory for fast branching */
256      pac_build_fast_branch();
257      if(pac_sweep_test(patterns) != SUCCESS)
258      {
259      returncode = FAILURE;
260      if(lm_error("Fast branching, random data, test failed.\n") != SUCCESS)
261      return(FAILURE);
262      }
263      /* Fill all of pattern memory with alternating ones and zeros data */
264      if(pac_fill_1and0(0, pac(current_lane).num_patterns, NOP_MASK) != SUCCESS)
265      {
266      (void)lm_error("Could not perform branch test.\n");
267      return(FAILURE);
268      }
269      /* Set up pattern memory for fast branching */
270      pac_build_fast_branch();
271      if(pac_sweep_test(patterns) != SUCCESS)
272      {
273      (void)lm_error("Fast branching, 1/0 data, test failed.\n");
274      return(FAILURE);
275      }
276      return(returncode);
277      }
278      }
279      }
280      /*
281      *   int pac_single_play()
282      *
283      *   INPUT: none
284      *   OUTPUT: return code = SUCCESS or FAILURE
285      *   DESCRIPTION:
286      */
287      int
288      pac_single_play()
289      {
290      int sample_width;
291      int first_pattern;
292      int patterns;
293      int play_time;
294      int period;
295
296      if(diag_clear_errors() != SUCCESS)
297      return(FAILURE);
298
299      sample_width = 16;
300      diag_get_long(&(long)sample_width, "sample width", 01, 2551);
301
302      first_pattern = 0;
303      diag_get_long(&(long)first_pattern, "first pattern number", 01,
304      (long)(pac(current_lane).num_patterns - BLOCK_SIZE));
305
306      patterns = pac(current_lane).num_patterns;
307      diag_get_long(&(long)patterns, "number of patterns", 01, (long)patterns);
308
309      if(build_pattern_control(first_pattern, patterns, STOP_MODE) != SUCCESS)
310      {
311      (void)lm_error("Unable to build pattern control.\n");
312      return(FAILURE);
313      }
314      /* Insert PFI control in last pattern to generate sample */
315      *((u_long *) (pac(current_lane).lane_offset + BANK_2 * 4 * (first_pattern +
316      patterns - 1))) |= 0x7 << 4;
317      sample_width = sample_width;
318      /* Prepare for pattern play and compute a conservative timeout */
319      if(pac_pre_play() == 0)
320      {
321      (void)lm_error("Pattern play preparation failed.\n");
322      return(FAILURE);
323      }
324
325      period = tny_measure_period();
326      setup_good_sample(period);
327      if(pac_timed_play(period / 1000, patterns, &play_time)
328      != SUCCESS)
329      {
330      (void)lm_error("Timed play fails.\n");
331      return(FAILURE);
332      }
333      (void)lm_message("Play lasted %dms +/- %dms.\n", play_time, TIMER_RES);
334      return(SUCCESS);
335      }
336      }
337      /*
338      *   int pac_loop_no_sample()
339      *
340      *   INPUT: none
341      *   OUTPUT: return code = SUCCESS or FAILURE
342      *   DESCRIPTION:
343      */
344      int
345      pac_loop_no_sample()
346      {
347      int first_pattern;
348      int patterns;
349      int period;
350
351      if(diag_clear_errors() != SUCCESS)
352      return(FAILURE);
353
354      /* Set sample width to 16 */
355      tnypr->sample_width = 16;
356
357      first_pattern = 0;
358      diag_get_long(&(long)first_pattern, "first pattern number", 01,
359      (long)(pac(current_lane).num_patterns - BLOCK_SIZE));
360

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_menu_diag.c

DATE 5/23/89
TIME 4:41:20 pm

PAGE #
4/66

```

LINE # SOURCE TEXT
361 patterns = pac(current_lane).num_patterns;
362 diag_get_long(&(long)patterns, "Number of patterns", 41, (long)patterns);
363
364 if(build_pattern_control(first_pattern, patterns, LOOP_MODE) != SUCCESS)
365 {
366     (void)lm_error("Unable to build pattern control.\n");
367     return(FAILURE);
368 }
369 /* Prepare for pattern play and compute a conservative timeout */
370 if(pac_pre_play() == 0)
371 {
372     (void)lm_error("Pattern play preparation failed.\n");
373     return(FAILURE);
374 }
375
376 period = tmg_measure_period();
377 setup_good_sample(period);
378
379 (void)lm_message("Initiating looping play...\n");
380 (void)lm_message("(Hit key to abort)\n");
381 if(tmg_initiate_play() != SUCCESS)
382 {
383     (void)lm_error("Unable to initiate play.\n");
384     pac_play_cleanup();
385     return(FAILURE);
386 }
387 /* Wait for key hit */
388 (void)lm_get_key();
389 Clear_key_buf();
390 if(Get_bp_error() == Lane_code(current_lane))
391 {
392     (void)lm_error("Error line asserted on backplane.\n");
393     report_bp_error();
394     return(FAILURE);
395 }
396 else
397 {
398     if(pac_abort_play() != SUCCESS)
399     {
400         (void)lm_error("Could not abort play.\n");
401         return(FAILURE);
402     }
403     else
404     {
405         (void)lm_message("Looping aborted.\n");
406         return(SUCCESS);
407     }
408 }
409 }
410
411 /*
412  * int pac_link_mem_test()
413  *
414  * INPUT: none
415  * OUTPUT: return code = SUCCESS or FAILURE
416  * DESCRIPTION: Tests link table memory. The
417  * link table is a 16-bit memory on 32-bit
418  * boundaries.
419  */
420 int
421 pac_link_mem_test()
422 {
423     if(pac_link_memtest() != SUCCESS)
424         return(FAILURE);
425     return(SUCCESS);
426 }
427
428 /*
429  * int pac_pat_mem_test(&status, &info)
430  *
431  * INPUT: &status = address of status word
432  *        &info = address of test info; in this case, PAM number
433  * OUTPUT: return code = SUCCESS or FAILURE
434  * DESCRIPTION: Tests pattern memory on specified PAM
435  */
436 int
437 pac_pat_mem_test(status, info)
438 int *status;
439 char *info;
440 {
441     if(pac_stack_pams(current_lane) != SUCCESS)
442     {
443         (void)lm_error("Pattern memory configuration error.\n");
444         return(FAILURE);
445     }
446
447     if(diag_clear_errors() != SUCCESS)
448         return(FAILURE);
449
450     return(pac_test_a_pam((int)('0' - '0')));
451 }
452
453 /*
454  * int pac_freq_test()
455  *
456  * INPUT: none
457  * OUTPUT: return code = SUCCESS or FAILURE
458  * DESCRIPTION:
459  */
460 int
461 pac_freq_test()
462 {
463     register int pam_no;
464     register int first_pattern;
465     register int num_patterns;
466     int returncode = SUCCESS;
467
468     if(pac_stack_pams(current_lane) != SUCCESS)
469     {
470         (void)lm_error("Pattern memory configuration error.\n");
471         return(FAILURE);
472     }
473
474     if(diag_clear_errors() != SUCCESS)
475         return(FAILURE);
476
477     for(pam_no = 0; pam_no < pac(current_lane).num_pams; pam_no++)
478     {
479         first_pattern = pac_get_first_pattern_no(current_lane, pam_no);
480         num_patterns = pac(current_lane).pam_size[pam_no];

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_menu_diag.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:20 pm | 5/67 |

```

LINE #          SOURCE TEXT
481
482 /* Fill pattern memory with random data */
483 if(pac_fill_random(first_pattern, num_patterns, MOP_MASK, SEED) != SUCCESS)
484 {
485     returncode = FAILURE;
486     if(!m_error("Could not perform frequency test on PAM ad.\n", pam_bo)
487         != SUCCESS)
488         return(FAILURE);
489     else
490         continue;
491 }
492 /* Set up pattern memory to branch through pattern memory */
493 if(build_pattern_control(first_pattern, num_patterns, STOP_MODE) != SUCCESS)
494 {
495     returncode = FAILURE;
496     if(!m_error("Could not perform frequency test on PAM ad.\n", pam_bo)
497         != SUCCESS)
498         return(FAILURE);
499     else
500         continue;
501 }
502 if(pac_sweep_test(num_patterns) != SUCCESS)
503 {
504     returncode = FAILURE;
505     if(!m_error("Frequency sweep test failed using PAM ad.\n", pam_bo)
506         != SUCCESS)
507         return(FAILURE);
508     else
509         continue;
510 }
511 }
512 return(returncode);
513 }
514
515 /*
516  * int pac_fill_mem_random()
517  *
518  * INPUT: none
519  * OUTPUT: return code = SUCCESS or FAILURE
520  * DESCRIPTION: fills entire pattern memory with
521  * a pseudorandom sequence.
522  */
523 int
524 pac_fill_mem_random()
525 {
526     if(pac_fill_random(0, pac(current_lane).num_patterns, "01, SEED) != SUCCESS)
527     {
528         (void)m_error("Could not fill pattern memory.\n");
529         return(FAILURE);
530     }
531     return(SUCCESS);
532 }
533
534 /*
535  * int pac_fill_mem_counting()
536  *
537  * INPUT: none
538  * OUTPUT: return code = SUCCESS or FAILURE
539  * DESCRIPTION: fills entire pattern memory with
540  * counting data.
541  */
542 int
543 pac_fill_mem_counting()
544 {
545     if(pac_fill_counting(0, pac(current_lane).num_patterns, "01) != SUCCESS)
546     {
547         (void)m_error("Could not fill pattern memory.\n");
548         return(FAILURE);
549     }
550     return(SUCCESS);
551 }
552
553 /*
554  * int pac_fill_mem_walking()
555  *
556  * INPUT: none
557  * OUTPUT: return code = SUCCESS or FAILURE
558  * DESCRIPTION: fills entire pattern memory with
559  * walking ones data.
560  */
561 int
562 pac_fill_mem_walking()
563 {
564     if(pac_fill_walking(0, pac(current_lane).num_patterns, "01, 1) != SUCCESS)
565     {
566         (void)m_error("Could not fill pattern memory.\n");
567         return(FAILURE);
568     }
569     return(SUCCESS);
570 }
571
572 /*
573  * int pac_fill_mem_walking0()
574  *
575  * INPUT: none
576  * OUTPUT: return code = SUCCESS or FAILURE
577  * DESCRIPTION: fills entire pattern memory with
578  * walking zeros data.
579  */
580 int
581 pac_fill_mem_walking0()
582 {
583     if(pac_fill_walking(0, pac(current_lane).num_patterns, "01, 0) != SUCCESS)
584     {
585         (void)m_error("Could not fill pattern memory.\n");
586         return(FAILURE);
587     }
588     return(SUCCESS);
589 }
590
591 /*
592  * int pac_fill_mem_land0()
593  *
594  * INPUT: none
595  * OUTPUT: return code = SUCCESS or FAILURE
596  * DESCRIPTION: fills entire pattern memory with
597  * alternating ones and zeros data.
598  */
599 int
600 pac_fill_mem_land0()
601 {
602     if(pac_fill_land0(0, pac(current_lane).num_patterns, "01) != SUCCESS)

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_menu_diag.c

DATE 5/23/89 PAGE #
TIME 4:41:20 pm 6/68

```

LINE # SOURCE TEXT
601 {
602     (void)lm_error("Could not fill pattern memory.\n");
603     return(FAILURE);
604 }
605 return(SUCCESS);
606 }
607
608
609 /*
610  * int pac_detect_pams(istatus)
611  *
612  * INPUT: istatus = address of status word
613  * OUTPUT: return code = SUCCESS or FAILURE
614  * DESCRIPTION: Checks to see if PAMs can be tested
615  *             detected on current PAC. If not, tests are aborted.
616  */
617 int
618 pac_detect_pams(istatus)
619     u_long *istatus;
620 {
621     if(pac_get_num_pams(current_lase) != SUCCESS)
622     {
623         (void)lm_error("PAM detect test failed. Cannot perform further tests \
624 on PAC.\n");
625         STOP_TEST(istatus);
626         return(FAILURE);
627     }
628     return(SUCCESS);
629 }
630
631 /*
632  * int pac_check_pamid(istatus)
633  *
634  * INPUT: istatus = address of status word
635  * OUTPUT: return code = SUCCESS or FAILURE
636  * DESCRIPTION: Checks the PAM ID Proms. If any
637  *             fail checksum, tests are aborted.
638  */
639 int
640 pac_check_pamid(istatus)
641     u_long *istatus;
642 {
643     static char message[] =
644         "PAM ID Prom test failed. Cannot perform further tests on PAC.\n";
645     if(pac(current_lase).num_pams == 0)
646     {
647         (void)lm_error("No pattern memory in this lase.\n");
648         return FAILURE;
649     }
650     if(pam_idprom_test(current_lase) != SUCCESS)
651     {
652         (void)lm_error(message);
653         STOP_TEST(istatus);
654         return(FAILURE);
655     }
656     return(SUCCESS);
657 }
658
659
660
661
662 pac_strapping_test(istatus)
663     u_long *istatus;
664 {
665     if(pac_stack_pams(current_lase) != SUCCESS)
666     {
667         (void)lm_error("Pattern memory configuration error.\n");
668         STOP_TEST(istatus);
669         return(FAILURE);
670     }
671     return SUCCESS;
672 }
673
674
675 /*
676  * int pac_fill_mem_sero()
677  *
678  * INPUT: none
679  * OUTPUT: return code = SUCCESS or FAILURE
680  * DESCRIPTION: Clears pattern memory.
681  */
682 int
683 pac_fill_mem_sero()
684 {
685     /* Check to make sure that there are PAMs on the PAC */
686     if(pac(current_lase).num_pams == 0)
687     {
688         (void)lm_error("There are no PAMs on this PAC.\n");
689         return(FAILURE);
690     }
691     else
692     {
693         (void)pac_clear_pat_mem(current_lase);
694         return(SUCCESS);
695     }
696 }
697
698 /*
699  * int pac_sel_pat_clk()
700  *
701  * INPUT: none
702  * OUTPUT: return code = SUCCESS or FAILURE
703  * DESCRIPTION: Sets pattern clock frequency by getting
704  *             the desired clock period from the keyboard. Also allows
705  *             the user to select an external clock. The clock is not
706  *             turned on by the function.
707  */
708 int
709 pac_sel_pat_clk()
710 {
711     int user_input;
712     (void)lm_message("Enter 0 for external clock 0.\n");
713     (void)lm_message("enter 1 for external clock 1.\n");
714     (void)lm_message("or enter period for internal clock.\n");
715     (void)lm_message("(id <= period <= id)\n", PAC_MIN_PERIOD, PAC_MAX_PERIOD);
716     user_input = 401; /* 25 MHz */
717     diag_get_log(&(long)user_input, "value", 01, (long)PAC_MAX_PERIOD);
718     if(user_input == 0)
719     {
720         tmpptr->clock_select = EXT_CLOCK_0;
721         (void)lm_message("External clock 0 selected.\n");
722     }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_menu_diag.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:20 pm | 7/69 |

| LINE # | SOURCE TEXT |
|--------|--|
| 721 | } |
| 722 | else if(user_input == 1) |
| 723 | { |
| 724 | tmpstr->clock_select = EXT_CLOCK_1; |
| 725 | (void)lm_message("External clock 1 selected.\n"); |
| 726 | } |
| 727 | else |
| 728 | { |
| 729 | if(pac_set_pattern_clock(user_input) != SUCCESS) |
| 730 | { |
| 731 | (void)lm_error("Unable to set internal clock frequency.\n"); |
| 732 | return(FAILURE); |
| 733 | } |
| 734 | else |
| 735 | { |
| 736 | /* Make sure PAC clock speed register set properly */ |
| 737 | (void)pac_pre_play(); |
| 738 | } |
| 739 | } |
| 740 | return(SUCCESS); |
| 741 | } |
| 742 | |
| 743 | |
| 744 | /* |
| 745 | int pac_idprom_test() |
| 746 | { |
| 747 | INPUT: none |
| 748 | OUTPUT: return code = SUCCESS or FAILURE |
| 749 | DESCRIPTION: Performs checksum test on PAC ID PROM. |
| 750 | The function returns SUCCESS if the checksum is |
| 751 | correct, and returns FAILURE if the checksum test fails. |
| 752 | } |
| 753 | int |
| 754 | pac_idprom_test() |
| 755 | { |
| 756 | int tmp; |
| 757 | if((tmp = id_checksum((u_char *) (pac[current_lane].lane_offset + |
| 758 | PAC_ID + 3))) != ID_CHECKSUM_GOOD) |
| 759 | { |
| 760 | if(lm_error("PAC ID PROM checksum. Expected = 002x. Actual = 002x.\n", |
| 761 | ID_CHECKSUM_GOOD, tmp) != SUCCESS) |
| 762 | return(FAILURE); |
| 763 | } |
| 764 | return(SUCCESS); |
| 765 | } |
| 766 | |
| 767 | /* |
| 768 | int pac_reg_test() |
| 769 | { |
| 770 | INPUT: none |
| 771 | OUTPUT: return code = SUCCESS or FAILURE |
| 772 | DESCRIPTION: Performs register test on all PAC read/write |
| 773 | registers. Returns SUCCESS or FAILURE. |
| 774 | } |
| 775 | int |
| 776 | pac_reg_test() |
| 777 | { |
| 778 | u_long addr; |
| 779 | u_long state; |
| 780 | int returncode = SUCCESS; |
| 781 | |
| 782 | /* Test the PAC Configuration Register */ |
| 783 | addr = pac[current_lane].lane_offset + CONFIG_OFFSET; |
| 784 | state = Read_long(addr); /* Save the state of the register */ |
| 785 | if(location_test(addr, 0, 8) != SUCCESS) |
| 786 | { |
| 787 | returncode = FAILURE; |
| 788 | if(lm_error("PAC Configuration Register test fails.\n") != SUCCESS) |
| 789 | return(FAILURE); |
| 790 | } |
| 791 | Write_long(addr, state); /* Return register to former state */ |
| 792 | |
| 793 | /* Test the PAC Clock Speed Register */ |
| 794 | addr = pac[current_lane].lane_offset + CLOCK_OFFSET; |
| 795 | state = Read_long(addr); /* Save the state of the register */ |
| 796 | if(location_test(addr, 0, 1) != SUCCESS) |
| 797 | { |
| 798 | returncode = FAILURE; |
| 799 | if(lm_error("PAC Clock Speed Register test fails.\n") != SUCCESS) |
| 800 | return(FAILURE); |
| 801 | } |
| 802 | Write_long(addr, state); /* Return register to former state */ |
| 803 | |
| 804 | /* Test the PAC Branch Address Register */ |
| 805 | addr = pac[current_lane].lane_offset + BRANCH_OFFSET; |
| 806 | state = Read_long(addr); /* Save the state of the register */ |
| 807 | if(location_test(addr, 0, 16) != SUCCESS) |
| 808 | { |
| 809 | returncode = FAILURE; |
| 810 | if(lm_error("PAC Branch Address Register test fails.\n") != SUCCESS) |
| 811 | return(FAILURE); |
| 812 | } |
| 813 | Write_long(addr, state); /* Return register to former state */ |
| 814 | |
| 815 | return(returncode); |
| 816 | } |
| 817 | |
| 818 | int |
| 819 | pac_config_report() |
| 820 | { |
| 821 | int i; |
| 822 | |
| 823 | if(pac_stack_pams(current_lane) != SUCCESS) |
| 824 | { |
| 825 | (void)lm_error("PAC configuration error.\n"); |
| 826 | return(FAILURE); |
| 827 | } |
| 828 | else |
| 829 | { |
| 830 | (void)lm_message("PAC configuration successful.\n"); |
| 831 | (void)lm_message("There are a total of %d PAMs.\n", |
| 832 | pac[current_lane].num_pams); |
| 833 | for(i=0, i < pac[current_lane].num_pams; i++) |
| 834 | { |
| 835 | (void)lm_message(" PAM %d is a %dk PAM.\n", i, |
| 836 | pac[current_lane].pam_size[i] >> 10); |
| 837 | } |
| 838 | (void)lm_message("There are a total of %dk patterns (%d blocks).\n", |
| 839 | pac[current_lane].num_patterns >> 10, pac[current_lane].num_blocks); |
| 840 | } |
| 841 | return(SUCCESS); |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_menu_diag.c

DATE 5/23/89
TIME 4:41:20 pm

PAGE #
8/70

```

LINE # SOURCE TEXT
841 }
842 }
843
844 int
845 pac_check_errors(lane, ptrfcn)
846 int lane;
847 int (*ptrfcn)();
848 {
849     int returncode = SUCCESS;
850
851     if((pacptr[lane]-->refresh_error) != 0)
852     {
853         returncode = FAILURE;
854         (void)ptrfcn("Pattern Controller Refresh error.\n");
855     }
856     if((pacptr[lane]-->request_error) != 0)
857     {
858         returncode = FAILURE;
859         (void)ptrfcn("Pattern Controller Request Control Machine error.\n");
860     }
861     if((pacptr[lane]-->pattern_error) != 0)
862     {
863         returncode = FAILURE;
864         (void)ptrfcn("Pattern Control Machine error.\n");
865     }
866     if((pacptr[lane]-->parity_error) != 0)
867     {
868         returncode = FAILURE;
869         (void)ptrfcn("Pattern Controller pattern control word parity error.\n");
870         block_number = 0;
871         block_offset = 0;
872         pacptr[lane]-->branch_address;
873         pacptr[lane]-->block_offset;
874     }
875     if((pacptr[lane]-->high_word_parity_error) != 0)
876     {
877         returncode = FAILURE;
878         (void)ptrfcn("Pattern Controller High Word parity error.\n");
879     }
880     if((pacptr[lane]-->low_word_parity_error) != 0)
881     {
882         returncode = FAILURE;
883         (void)ptrfcn("Pattern Controller Low Word parity error.\n");
884     }
885     if((pacptr[lane]-->low_word_parity_error) != 0)
886     {
887         if((pacptr[lane]-->high_word_parity_error) != 0)
888         {
889             (void)ptrfcn("Pattern Controller parity error address = 108X.\n");
890             Pac_parity_address(lane);
891             return(returncode);
892         }
893     }
894
895     int
896     pac_display_errors()
897     {
898         if(pac_check_errors(current_lane, lm_message) == SUCCESS)
899             (void)lm_message("No Pattern Controller error conditions present.\n");
900         return(SUCCESS);
901     }
902
903     /*
904     int pac_loop_no_pams()
905     * INPUT: none
906     * OUTPUT: return code = SUCCESS or FAILURE
907     * DESCRIPTION:
908     */
909     int
910     pac_loop_no_pams()
911     {
912         int timeout;
913         int period;
914
915         /* Check to make sure that there are no PAMs on the PAC */
916         if(pac(current_lane).num_pams != 0)
917         {
918             (void)lm_error("There are PAMs on this PAC.\n");
919             return(FAILURE);
920         }
921
922         Clear_pac_errors(current_lane);
923
924         pacptr[current_lane]-->low_word_parity = SET_ODD_PARITY;
925         tmgptr-->sample_width = 16;
926
927         /* Prepare for pattern play and compute a conservative timeout */
928         if((timeout = pac_pre_play()) == 0)
929         {
930             (void)lm_error("Pattern play preparation failed.\n");
931             return(FAILURE);
932         }
933         period = tmg_measure_period();
934         setup_good_sample(period);
935         (void)lm_message("Initiating looping play...\n");
936         (void)lm_message("Hit key to abort.\n");
937         while((Get_bp_error() != Lane_code(current_lane)) && (lm_check_key() == 0x0))
938         {
939             pacptr[current_lane]-->branch_address = 0;
940             if(pac_play(timeout) != SUCCESS)
941             {
942                 pac_play_cleanup();
943                 break;
944             }
945         }
946         Clear_key_buf();
947         if(Get_bp_error() == Lane_code(current_lane))
948         {
949             (void)lm_error("Error line asserted on backplane.\n");
950             (void)report_bp_error();
951         }
952         else
953             (void)lm_message("Looping aborted.\n");
954         return(SUCCESS);
955     }
956
957     /*
958     int pac_loop_with_sample()
959     * INPUT: none
960     * OUTPUT: return code = SUCCESS or FAILURE
961     * DESCRIPTION:
962     */

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_menu_diag.c

DATE 5/23/89
TIME 4:41:20 pm

PAGE #
9/71

```

LINE # SOURCE TEXT
961 int
962 pac_loop_with_sample()
963 {
964     long sample_width;
965     int first_pattern;
966     int first_block;
967     int patterns;
968     int timeout;
969     int period;
970
971     if(diag_clear_errors() != SUCCESS)
972         return(FAILURE);
973
974     sample_width = 16;
975     diag_get_log(&sample_width, "sample width", 01, 2551);
976     first_pattern = 0;
977     diag_get_log(&(long)first_pattern, "first pattern number", 01,
978     (long)(pac(current_lane).sum_patterns - BLOCK_SIZE));
979     patterns = pac(current_lane).sum_patterns;
980     diag_get_log(&(long)patterns, "number of patterns", 41, (long)patterns);
981
982     if(build_patterns_control(first_pattern, patterns, STOP_MODE) != SUCCESS)
983     {
984         (void)lm_error("Unable to build patterns control.\n");
985         return(FAILURE);
986     }
987     /* Insert PEL control in last pattern to generate sample */
988     *(u_long *)(&pac(current_lane).lane_offset + BANK_2 + 4 * (first_pattern +
989     patterns - 1)) |= 0x7 << 4;
990     trigger->sample_width = sample_width;
991     first_block = first_pattern/BLOCK_SIZE;
992
993     /* Prepare for patterns play and compute a conservative timeout */
994     if((timeout = pac_pre_play()) == 0)
995     {
996         (void)lm_error("Patterns play preparation failed.\n");
997         return(FAILURE);
998     }
999     period = tmg_measure_period();
1000     setup_good_sample(period);
1001     (void)lm_message("Initiating looping play...\n");
1002     (void)lm_message("(Hit key to abort)\n");
1003     while((Get_bp_error() != Lane_code(current_lane)) && (lm_check_key() == 0x0))
1004     {
1005         pacptr(current_lane)->branch_address = first_block;
1006         if(pac_play(timeout) != SUCCESS)
1007         {
1008             pac_play_cleanup();
1009             break;
1010         }
1011     }
1012     Clear_key_buf();
1013     if(Get_bp_error() == Lane_code(current_lane))
1014     {
1015         (void)lm_error("Error line asserted on backplane.\n");
1016         (void)report_bp_error();
1017     }
1018     else
1019     {
1020         (void)lm_message("Looping aborted.\n");
1021         return(SUCCESS);
1022     }
1023
1024     /*
1025     * int pac_sig_analysis()
1026     *
1027     * INPUT: none
1028     * OUTPUT: return code = SUCCESS or FAILURE
1029     * DESCRIPTION: Sets lane in signature analysis
1030     * mode, playing random data through entire memory.
1031     * The trigger pin is set in the first pattern only.
1032     */
1033     int
1034     pac_sig_analysis()
1035     {
1036         if(diag_clear_errors() != SUCCESS)
1037             return(FAILURE);
1038
1039         if(pac_fill_random(0, pac(current_lane).sum_patterns, "0x0801, SEED")
1040         != SUCCESS)
1041         {
1042             (void)lm_error("Could not fill pattern memory.\n");
1043             return(FAILURE);
1044         }
1045         if(build_patterns_control(0, pac(current_lane).sum_patterns, LOOP_MODE)
1046         != SUCCESS)
1047         {
1048             (void)lm_error("Unable to build pattern control.\n");
1049             return(FAILURE);
1050         }
1051         /* Set trigger bit in first location of pattern memory */
1052         *(u_long *)(&pac(current_lane).lane_offset + BANK_2) |= 0x0801;
1053         /* Prepare for patterns play and compute a conservative timeout */
1054         if(pac_pre_play() == 0)
1055         {
1056             (void)lm_error("Patterns play preparation failed.\n");
1057             return(FAILURE);
1058         }
1059
1060         pacptr(current_lane)->branch_address = 0;
1061         (void)lm_message("Initiating signature analysis...\n");
1062         (void)lm_message("(Hit key to abort)\n");
1063         if(tmg_initiate_play() != SUCCESS)
1064         {
1065             (void)lm_error("Unable to initiate play.\n");
1066             pac_play_cleanup();
1067             return(FAILURE);
1068         }
1069         /* Wait for key hit */
1070         while(lm_check_key() == 0x0)
1071         {
1072             Clear_key_buf();
1073             if(Get_bp_error() == Lane_code(current_lane))
1074             {
1075                 (void)lm_error("Error line asserted on backplane.\n");
1076                 (void)report_bp_error();
1077                 return(FAILURE);
1078             }
1079             else
1080             {
1081                 if(pac_abort_play() != SUCCESS)

```


| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pac_menu_diag.c | DATE 5/23/89 | PAGE # 10/72 |
|--|--|---|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 1081 | | { | | |
| 1082 | | {(void)lm_error("Could not abort play.\n"); | | |
| 1083 | | return(FAILURE); | | |
| 1084 | | } | | |
| 1085 | | else | | |
| 1086 | | { | | |
| 1087 | | {(void)lm_message("Signature analysis halted.\n"); | | |
| 1088 | | return(SUCCESS); | | |
| 1089 | | } | | |
| 1090 | | } | | |
| 1091 | | } | | |
| 1092 | | /* | | |
| 1093 | | int pac_pattern_bits_test() | | |
| 1094 | | /* | | |
| 1095 | | INPUT: none | | |
| 1096 | | OUTPUT: return code = SUCCESS or FAILURE | | |
| 1097 | | DESCRIPTION: | | |
| 1098 | | */ | | |
| 1099 | | int | | |
| 1100 | | pac_pattern_bits_test() | | |
| 1101 | | { | | |
| 1102 | | int pam_no; | | |
| 1103 | | int pels_found; | | |
| 1104 | | int pama_failed; | | |
| 1105 | | int pels_failed = 0; | | |
| 1106 | | if(pac_stack_pama(current_lase) != SUCCESS) | | |
| 1107 | | { | | |
| 1108 | | {(void)lm_error("Pattern memory configuration error.\n"); | | |
| 1109 | | return(FAILURE); | | |
| 1110 | | } | | |
| 1111 | | if(diag_clear_errors() != SUCCESS) | | |
| 1112 | | return(FAILURE); | | |
| 1113 | | /* See if there are any PELS in the lase */ | | |
| 1114 | | if((pels_found = pac_check_for_pels()) == 0) | | |
| 1115 | | { | | |
| 1116 | | {(void)lm_error("Can not test pattern bits (no Pin Electronics in lase).\n"); | | |
| 1117 | | return(FAILURE); | | |
| 1118 | | } | | |
| 1119 | | /* Test the PELS one at a time (need to set global PEL variable) */ | | |
| 1120 | | /* The first PEL which passes the test => the PAC passes and the */ | | |
| 1121 | | /* test is over */ | | |
| 1122 | | for(current_pel = 0; current_pel < NUMBER_OF_PELS; ++current_pel) | | |
| 1123 | | { | | |
| 1124 | | if(((1 << current_pel) & pels_found) == 0) /* current PEL not found */ | | |
| 1125 | | continue; | | |
| 1126 | | /* Test the pattern bits coming out of each pattern memory on the PAC */ | | |
| 1127 | | pama_failed = FALSE; | | |
| 1128 | | for(pam_no = 0; pam_no < pac(current_lase).sum_pama; ++pam_no) | | |
| 1129 | | { | | |
| 1130 | | if(pattern_bits_test(pam_no) != SUCCESS) | | |
| 1131 | | { | | |
| 1132 | | pama_failed = TRUE; | | |
| 1133 | | pels_failed = 1 << current_pel; | | |
| 1134 | | if(lm_error("Pattern bits test failed using Pin Electronics id, \n | | |
| 1135 | | Pattern Memory id.\n", current_pel, pam_no) != SUCCESS) | | |
| 1136 | | return(FAILURE); | | |
| 1137 | | } | | |
| 1138 | | if(pama_failed == FALSE) | | |
| 1139 | | { | | |
| 1140 | | if(pels_failed == 0) | | |
| 1141 | | return(SUCCESS); | | |
| 1142 | | else | | |
| 1143 | | {(void)lm_message("Pattern bits test passed using Pin Electronics id.\n", | | |
| 1144 | | current_pel); | | |
| 1145 | | } | | |
| 1146 | | /* If we got here, then at least one test failed for each PEL */ | | |
| 1147 | | if((pels_found == pels_failed)) | | |
| 1148 | | {(void)lm_error("Pattern bits test failed for all Pin Electronics in \n | | |
| 1149 | | lase.\n"); | | |
| 1150 | | else | | |
| 1151 | | {(void)lm_message("Pattern bits test passed at least once in lase.\n"); | | |
| 1152 | | return(FAILURE); | | |
| 1153 | | } | | |
| 1154 | | /* | | |
| 1155 | | int pac_pattern_bus_test() | | |
| 1156 | | /* | | |
| 1157 | | INPUT: none | | |
| 1158 | | OUTPUT: return code = SUCCESS or FAILURE | | |
| 1159 | | DESCRIPTION: | | |
| 1160 | | */ | | |
| 1161 | | int | | |
| 1162 | | pac_pattern_bus_test() | | |
| 1163 | | { | | |
| 1164 | | int first_pattern; | | |
| 1165 | | int pels_found; | | |
| 1166 | | int pama_failed; | | |
| 1167 | | int pels_failed = 0; | | |
| 1168 | | if(pac_stack_pama(current_lase) != SUCCESS) | | |
| 1169 | | { | | |
| 1170 | | {(void)lm_error("Pattern memory configuration error.\n"); | | |
| 1171 | | return(FAILURE); | | |
| 1172 | | } | | |
| 1173 | | if(diag_clear_errors() != SUCCESS) | | |
| 1174 | | return(FAILURE); | | |
| 1175 | | /* See if there are any PELS in the lase */ | | |
| 1176 | | if((pels_found = pac_check_for_pels()) == 0) | | |
| 1177 | | { | | |
| 1178 | | {(void)lm_error("Cannot test pattern bus (no Pin Electronics in lase).\n"); | | |
| 1179 | | return(FAILURE); | | |
| 1180 | | } | | |
| 1181 | | /* Test the PELS one at a time (need to set global PEL variable) */ | | |
| 1182 | | /* The first PEL which passes the test => the PAC passes and the */ | | |
| 1183 | | /* test is over */ | | |
| 1184 | | for(current_pel = 0; current_pel < NUMBER_OF_PELS; ++current_pel) | | |
| 1185 | | { | | |
| 1186 | | if(((1 << current_pel) & pels_found) == 0) /* current PEL not found */ | | |
| 1187 | | continue; | | |
| 1188 | | /* | | |
| 1189 | | int | | |
| 1190 | | pac_pattern_bits_test() | | |
| 1191 | | { | | |
| 1192 | | int pam_no; | | |
| 1193 | | int pels_found; | | |
| 1194 | | int pama_failed; | | |
| 1195 | | int pels_failed = 0; | | |
| 1196 | | if(pac_stack_pama(current_lase) != SUCCESS) | | |
| 1197 | | { | | |
| 1198 | | {(void)lm_error("Pattern memory configuration error.\n"); | | |
| 1199 | | return(FAILURE); | | |
| 1200 | | } | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_menu_diag.c

DATE 5/23/89 PAGE #
TIME 4:41:20 pm 11/73

```

1201  /* Test all of the pattern memory on the PAC is 128K chunks */
1202  pams_failed = FALSE;
1203  for(first_pattern = 0, first_pattern < pac[current_lane].num_patterns,
1204  first_pattern += PATTERNS_IN_128K)
1205  {
1206      if(patterns_bus_test(first_pattern, PATTERNS_IN_128K) != SUCCESS)
1207      {
1208          pams_failed = TRUE;
1209          pels_failed |= 1 << current_pel;
1210          if(lm_error("Pattern bus test failed using Pin Electronics td, Pattern \
1211 Memory td.\n", current_pel, pac_get_pam_number(first_pattern)) != SUCCESS)
1212              return(FAILURE);
1213      }
1214  }
1215  if(pams_failed == FALSE)
1216  {
1217      if(pels_failed == 0)
1218          return(SUCCESS);
1219      else
1220          (void)lm_message("Pattern bus test passed using Pin Electronics td.\n",
1221 current_pel);
1222  }
1223  /* If we got here, then at least one test failed for each PEL */
1224  if((pels_found == pels_failed))
1225      (void)lm_error("Pattern bus test failed for all Pin Electronics in \
1226 lane.\n");
1227  else
1228      (void)lm_message("Pattern bus test passed at least once in lane.\n");
1229  return(FAILURE);
1230 }
1231
1232 /*
1233  int pac_crc_test()
1234  *
1235  * INPUT: none
1236  * OUTPUT: returns code = SUCCESS or FAILURE
1237  * DESCRIPTION:
1238  */
1239 int
1240 pac_crc_test()
1241 {
1242     int first_pattern;
1243     int dabs_found;
1244     int pams_failed;
1245     int dabs_failed = 0;
1246
1247     if(pac_stack_pams(current_lane) != SUCCESS)
1248     {
1249         (void)lm_error("Pattern memory configuration error.\n");
1250         return(FAILURE);
1251     }
1252
1253     if(diag_clear_errors() != SUCCESS)
1254         return(FAILURE);
1255
1256     /* See if there are any Diagnostic Adapters in the lane */
1257     if((dabs_found = pac_check_for_diag_dabs(pac_check_for_pels())) == 0)
1258     {
1259         (void)lm_error("Cannot perform CRC (no Diagnostic Adapters in lane).\n");
1260         return(FAILURE);
1261     }
1262
1263     /* Test the PELs one at a time (need to set global PEL variable) */
1264     /* The first PEL which passes the test => the PAC passes and the */
1265     /* test is over */
1266     for(current_pel = 0, current_pel < NUMBER_OF_PELS, ++current_pel)
1267     {
1268         if(((1 << current_pel) & dabs_found) == 0) /* current DAB not found */
1269             continue;
1270
1271         /* Set up the PEL and the DAB for the CRC test (set global dabs_type) */
1272         dabs_type = DIAG_DAB;
1273         if(pel_crc_setup() != SUCCESS)
1274         {
1275             (void)lm_error("Unable to perform Pattern Controller CRC test with Pin \
1276 Electronics in slot td.\n", current_pel);
1277             dabs_failed |= 1 << current_pel;
1278             continue;
1279         }
1280         /* Test all of the pattern memory on the PAC is 128K chunks */
1281         pams_failed = FALSE;
1282         for(first_pattern = 0, first_pattern < pac[current_lane].num_patterns,
1283 first_pattern += PATTERNS_IN_128K)
1284         {
1285             if(pel_crc_test_128K(first_pattern) != SUCCESS)
1286             {
1287                 pams_failed = TRUE;
1288                 dabs_failed |= 1 << current_pel;
1289                 if(lm_error("CRC test failed using Pin Electronics td, Pattern \
1290 Memory td.\n", current_pel, pac_get_pam_number(first_pattern)) != SUCCESS)
1291                     return(FAILURE);
1292             }
1293         }
1294         if(pams_failed == FALSE)
1295         {
1296             if(dabs_failed == 0)
1297                 return(SUCCESS);
1298             else
1299                 (void)lm_message("CRC test passed using Pin Electronics in slot td.\n",
1300 current_pel);
1301         }
1302     }
1303     /* If we got here, then at least one test failed for each PEL */
1304     if((dabs_found == dabs_failed))
1305         (void)lm_error("CRC test failed for all Pin Electronics in lane.\n");
1306     else
1307         (void)lm_message("Pattern bus test passed at least once in lane.\n");
1308     return(FAILURE);
1309 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_util.c

DATE 5/23/89
TIME 4:41:22 pm

PAGE #
1/74

```

LINE # SOURCE TEXT
1  /* SCCS ID: pac_util.c rev 3.1, 4/24/89 at 07:49:27 */
2  /*
3  *   pac_util.c
4  *   *
5  *   Utility routines
6  *   used in PAC diagnostics
7  *   *
8  *   *****
9  *
10 #include <math.h>
11 #include <vrtx.h> /* for lm_delay and lm_time */
12 #include <common.h>
13 #include <mod_def.h>
14 #include <moduler_exta.h>
15 #include <tag.h>
16 #include <tag_def.h>
17 #include <tag_exta.h>
18 #include <tag_fun.h>
19 #include <pac.h>
20 #include <magic.h>
21 #include <pel.h>
22 #include <pac_def.h>
23 #include <pac_exta.h>
24 #include <id.h>
25
26
27
28 int
29 pac_check_for_pels()
30 {
31     int slot;
32     int pels_found = 0;
33
34     for(slot = 0; slot < NUMBER_OF_SLOTS; ++slot)
35         pels_found |= (probe_pel(current_lane, slot) == SUCCESS) ? (1 << slot) : 0;
36     return(pels_found);
37 }
38
39 int
40 pac_check_for_diag_dabs(pels_found)
41 int pels_found;
42 {
43     int pel = 0;
44     int diag_dabs = 0;
45
46     pels_found &= ~(0 << NUMBER_OF_PELS);
47     while(pels_found > 0)
48     {
49         if((pels_found & 1) != 0)
50         {
51             if(what_dab(current_lane, pel) == DIAG_DAB)
52                 diag_dabs |= 1 << pel;
53             ++pel;
54             pels_found >>= 1;
55         }
56         return(diag_dabs);
57     }
58 }
59
60 /*
61  *   int pac_pre_play()
62  *   *
63  *   INPUT: none
64  *   OUTPUT: returns conservative timeout in ms (0 -> FAILURE)
65  *   DESCRIPTION: Measures clock frequency and then
66  *   sets clock speed register on PAC. Computes and returns
67  *   conservative timeout based on clock period and max
68  *   patterns in enabled lanes.
69  *   *
70  */
71 int
72 pac_pre_play()
73 {
74     int pac_compute_playtime();
75     int enabled_lanes;
76     int clock_period;
77     int max_patterns;
78     int lane_no;
79
80     /* Find out which lanes have been enabled */
81     enabled_lanes = tsgptr->lane_enable;
82
83     /* Measure the clock period */
84     if((clock_period = tsg_measure_period() / 1000) == 0)
85     {
86         (void)lm_error("Unable to measure the clock period.\n");
87         return(0);
88     }
89     /* Set the PAC clock speed registers in each of the enabled and configured
90      * lanes. Figure out the greatest number of patterns in enabled lanes */
91     for(lane_no = 0; max_patterns = 0; lane_no < NUMBER_OF_LANES; lane_no++)
92     {
93         if((lane_code(lane_no) & enabled_lanes & configured_lanes) != 0)
94         {
95             pac_clock_speed(lane_no, clock_period);
96             if(pac(lane_no).sum_patterns > max_patterns)
97                 max_patterns = pac(lane_no).sum_patterns;
98         }
99     }
100     /* Return a timeout based on a worst case expected play time */
101     return(timeout(pac_compute_   time(max_patterns, clock_period)));
102 }
103
104 /*
105  *   int pac_play(timeout)
106  *   *
107  *   INPUT: timeout = pattern play timeout in ms
108  *   OUTPUT: returns SUCCESS or FAILURE
109  *   DESCRIPTION: Performs a pattern play.
110  *   *
111  */
112 int pac_play(timeout)
113 int timeout;
114 {
115     /* Check for errors on the backplane */
116     if((Get_bp_error() & tsgptr->lane_enable) != 0)
117     {
118         report_bp_error();
119         (void)lm_error("Unable to play due to backplane error(s).\n");
120         return(FAILURE);
121     }
122     if(tsg_play((long)timeout) != SUCCESS)

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_util.c

DATE 5/23/89
TIME 4:41:22 pm

PAGE #
2/75

```

121  {
122  (void)is_error("Pattern play fails with a timeout of %dms.\n", timeout);
123  pac_play_cleanup();
124  return(FAILURE);
125  }
126  return(SUCCESS);
127  }
128  }
129  }
130  /*
131  *
132  * INPUT: none
133  * OUTPUT: none
134  * DESCRIPTION: Initializes PAC information table and
135  * PAC pointers.
136  */
137  void
138  pac_info_init()
139  {
140  int lane_no;
141  for(lane_no = 0; lane_no < NUMBER_OF_LANES; lane_no++)
142  {
143  pac[lane_no].lane_offset = LANE_A_OFFSET + (lane_no * LANE_SIZE);
144  pac[lane_no].num_patts = 0;
145  pac[lane_no].exists = FALSE;
146  pac[lane_no].num_blocks = Host ? 0x0000 : 01;
147  pac[lane_no].num_patterns = Host ? 0x0000 : 01;
148  pacptr[lane_no] = (PAC *) (pac[lane_no].lane_offset + PAC_RDC_OFFSET);
149  }
150  configured_lanes = WORK;
151  }
152  }
153  }
154  /*
155  *
156  * INPUT: pattern = pattern number (0 counting)
157  *        spattern_word = address of pattern to be written
158  * OUTPUT: returns SUCCESS or FAILURE
159  * DESCRIPTION: Writes pattern word to pattern location. Returns
160  * FAILURE if pattern number is outside pattern memory.
161  */
162  int
163  pac_write_pattern(pattern, spattern_word)
164  int pattern;
165  PAT_WORD *spattern_word;
166  {
167  u_long location;
168  /* Check if pattern number is valid */
169  if((pattern < 0) || (pattern > (pac[current_lane].num_patterns - 1)))
170  return(FAILURE);
171  /* Compute base address */
172  location = pac[current_lane].lane_offset + (4 * pattern);
173  /* Write pattern word */
174  Write_long(location + BANK_0, spattern_word->bank[0]);
175  Write_long(location + BANK_1, spattern_word->bank[1]);
176  Write_long(location + BANK_2, spattern_word->bank[2]);
177  return(SUCCESS);
178  }
179  }
180  }
181  /*
182  *
183  * INPUT: pattern = pattern number (0 counting)
184  *        spattern_word = address of pattern to place read values
185  * OUTPUT: returns SUCCESS or FAILURE
186  * DESCRIPTION: Reads pattern word from pattern location. Returns
187  * FAILURE if pattern number is outside pattern memory.
188  */
189  int
190  pac_read_pattern(pattern, spattern_word)
191  int pattern;
192  PAT_WORD *spattern_word;
193  {
194  u_long location;
195  /* Check if pattern number is valid */
196  if((pattern < 0) || (pattern > (pac[current_lane].num_patterns - 1)))
197  return(FAILURE);
198  /* Compute base address */
199  location = pac[current_lane].lane_offset + (4 * pattern);
200  /* Read pattern word */
201  spattern_word->bank[0] = Read_long(location + BANK_0);
202  spattern_word->bank[1] = Read_long(location + BANK_1);
203  spattern_word->bank[2] = Read_long(location + BANK_2);
204  return(SUCCESS);
205  }
206  }
207  }
208  /*
209  *
210  * INPUT: clock_period = pattern clock period in ns
211  *        patterns = number of patterns to play time before play times out
212  *        play_time = address of play_time
213  * OUTPUT: play_time = time of pattern play in ns (5ms resolution)
214  *        returns SUCCESS or FAILURE
215  * DESCRIPTION: Performs pattern play to selected lanes. Both
216  * the TMC and the PAC must be set up.
217  */
218  int
219  pac_timed_play(clock_period, patterns, play_time)
220  int clock_period;
221  int patterns;
222  int *play_time;
223  {
224  int pac_compute_playtime();
225  int start;
226  int expected_time;
227  int play_result;
228  *play_time = 0;
229  /* Check for errors on the backplane */
230  if((Get_bp_error() & tmcptr->lane_enable) != 0)
231  {
232  report_bp_error();
233  (void)is_error("Unable to play due to backplane error(s).\n");
234  }
235  }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_util.c

\$

DATE 5/23/89
TIME 4:41:22 pm

PAGE #
3/76

```

LINE # SOURCE TEXT
241     return(FAILURE);
242 }
243
244     expected_time = (long)pac_compute_playtime(patterns, clock_period);
245
246     start = lm_time();           /* Sync time */
247     while(start == lm_time())    /* and */
248     {                           /* then */
249         start = lm_time();       /* Record current time */
250     }
251     play_result = tmg_play((long)Timeout(expected_time));
252     play_time = lm_time() - start; /* Record play time */
253     if(play_result != SUCCESS)
254     {
255         (void)lm_error("tmg_play fails.\n");
256         pac_play_cleanup();
257         return(FAILURE);
258     }
259     /* Check to see if there is an error on the backplane */
260     if(report_bp_error() != 0)
261     {
262         (void)lm_error("Timed pattern play caused backplane error.\n");
263         return(FAILURE);
264     }
265     /* Check to see if actual time was close to expected time */
266     if((play_time < (expected_time - TIMER_RES)) ||
267         (play_time > (expected_time + TIMER_RES)))
268     {
269         (void)lm_error("Play timing error: Expected %d ms. Actual %d ms.\n",
270             expected_time, play_time);
271         return(FAILURE);
272     }
273     return(SUCCESS);
274 }
275
276 void pac_play_cleanup()
277 {
278     INPUT: none
279     OUTPUT: none
280     DESCRIPTION: Performs pattern play cleanup after a failed play.
281 }
282
283 void pac_play_cleanup()
284 {
285     if(bp_mode() == PLAY_MODE)
286     {
287         (void)lm_message("Backplane is still in play mode.\n");
288         (void)lm_message("Attempting to abort play.\n");
289         if(pac_abort_play() != SUCCESS)
290         {
291             (void)lm_message("Abort failed. Forcing backplane into access mode.\n");
292             tmgptr->abort_pattern_play = 1;
293             (void)pac_clock_off();
294             (void)pac_clock_on();
295             lm_delay(5);
296             (void)pac_clock_off();
297             if(bp_mode() == PLAY_MODE)
298             {
299                 (void)lm_message("Unable to force backplane into access mode.\n");
300             }
301         }
302     }
303 }
304
305 void pac_set_first_block(lanes, first_block)
306 {
307     INPUT: lanes = desired lanes for pattern play
308           first_block = block number of first pattern
309     OUTPUT: none
310     DESCRIPTION: Sets the branch address register to the
311                desired block number in each of the desired lanes. All
312                desired lanes must be configured.
313 }
314
315 void pac_set_first_block(lanes, first_block)
316 {
317     int lanes;
318     int first_block;
319     {
320         int lane_no;
321         for(lane_no = 0; lane_no < NUMBER_OF_LANES; lane_no++)
322         {
323             if((Lane_code(lane_no) & lanes & configured_lanes) != 0)
324             {
325                 pacptr(lane_no)->branch_address = first_block;
326             }
327         }
328     }
329 }
330
331 int pac_abort_play()
332 {
333     INPUT: none
334     OUTPUT: returns SUCCESS or FAILURE
335     DESCRIPTION: Aborts pattern play by asserting and then removing
336                the error line.
337 }
338
339 int pac_abort_play()
340 {
341     int start;
342     start = lm_time();
343     tmgptr->backplane_error = 1; /* Assert error line */
344     while(tmgptr->clock_on)      /* Wait for clocks to shut off */
345     {
346         if((lm_time() - start) > 5) /* 5ms timeout */
347         {
348             tmgptr->backplane_error = 0; /* Deassert error line */
349             (void)lm_error("Could not halt play - clocks are on.\n");
350             return(FAILURE);
351         }
352     }
353     tmgptr->backplane_error = 0; /* Deassert error line */
354     /* Flush error signal out of pipeline */
355     if(pac_clock_off() != SUCCESS) /* Turn off clock */
356     {
357         return(FAILURE);
358     }
359     if(pac_clock_on() != SUCCESS) /* Turn on clock */
360     {
361         return(FAILURE);
362     }
363     if(pac_clock_off() != SUCCESS) /* Turn off clock */
364     {
365         return(FAILURE);
366     }
367     if(bp_mode() == PLAY_MODE)
368     {
369         (void)lm_message("Unable to abort play.\n");
370         (void)lm_message("Backplane is still in play mode.\n");
371         return(FAILURE);
372     }
373 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_util.c

DATE 5/23/89
TIME 4:41:22 pm

PAGE #
4/77

```

LINE # SOURCE TEXT
361 }
362 return(SUCCESS);
363 }
364
365 /*
366  * int pac_clock_on()
367  *
368  * INPUT: none
369  * OUTPUT: returns SUCCESS or FAILURE
370  * DESCRIPTION: Turns on pattern clock.
371  */
372 int
373 pac_clock_on()
374 {
375     int start;
376
377     tmptr->clock_sync_clear1 = 1;
378     tmptr->clock_enable = 1;
379     start = lm_time();
380     while(!tmptr->clock_on)
381     {
382         if((lm_time() - start) > 5) /* 5ms timeout */
383         {
384             (void)lm_error("Unable to turn on clock.\n");
385             return(FAILURE);
386         }
387     }
388     return(SUCCESS);
389 }
390
391 /*
392  * int pac_clock_off()
393  *
394  * INPUT: none
395  * OUTPUT: returns SUCCESS or FAILURE
396  * DESCRIPTION: Turns off pattern clock.
397  */
398 int
399 pac_clock_off()
400 {
401     int start;
402
403     tmptr->clock_enable = 0;
404     start = lm_time();
405     while(tmptr->clock_off)
406     {
407         if((lm_time() - start) > 5) /* 5ms timeout */
408         {
409             (void)lm_error("Unable to turn off clock.\n");
410             return(FAILURE);
411         }
412     }
413     tmptr->clock_sync_clear1 = 0;
414     return(SUCCESS);
415 }
416
417 /*
418  * int pac_set_pattern_clock(period)
419  *
420  * INPUT: period = clock period in nanoseconds
421  * OUTPUT: returns SUCCESS or FAILURE
422  * DESCRIPTION: Sets pattern clock to period specified.
423  * This routine computes the values for n, k, and clock
424  * select register setting to give the best approximation
425  * of the desired clock period. The clock period provided
426  * is always equal to or greater than that requested.
427  * The actual clock period is related to n, k as follows:
428  * period = (k / n) * T_REF, where
429  * k = 1, 2, ..., 256 or k = 2, 4, ..., 512,
430  * n = 128, 129, ..., 256, and
431  * T_REF = 256 / 30 MHz = PLL reference period.
432  * Returns: FAILURE if period out of range, else SUCCESS
433  */
434 int
435 pac_set_pattern_clock(period)
436 {
437     int period;
438     double error;
439     double besterror;
440     double kprime;
441     int tmpk;
442     int n;
443     int save_k;
444     int save_n;
445
446     if((period > PAC_MAX_PERIOD) || (period < PAC_MIN_PERIOD))
447     {
448         return(FAILURE);
449     }
450     if(pac_clock_off() != SUCCESS)
451     {
452         return(FAILURE);
453     }
454     for(besterror = 1.0, n = N_MIN, k = N_MAX, k++)
455     {
456         kprime = n * period / 8533.3333; /* 8533.3333 = T_REF in ns */
457         if((tmpk = (int)ceil(kprime)) > 512) /* see if out of range */
458             continue; /* if so, ignore it */
459         if(tmpk > 256) /* see if above 256 */
460             continue; /* see if odd */
461         if(tmpk & 1) /* make even */
462             tmpk++;
463         if((error = (((double)tmpk - kprime) / kprime)) < besterror)
464         {
465             save_n = n;
466             save_k = tmpk;
467             besterror = error;
468         }
469     }
470     tmptr->pll_rate = 256 - save_n + 1; /* put in reg */
471     if(save_k == 1)
472     {
473         tmptr->pll_divisor = 255; /* special case */
474         tmptr->clock_select = 0; /* select div by 2 */
475     }
476     else
477     {
478         if(save_k > 256)
479         {
480             tmptr->pll_divisor = 256 - (save_k >> 1) + 1; /* divide k by 2 */
481             tmptr->clock_select = 2; /* select div by 4k */
482         }
483         else
484         {
485             tmptr->pll_divisor = 256 - (save_k >> 1) + 1; /* divide k by 2 */
486             tmptr->clock_select = 2; /* select div by 4k */
487         }
488     }
489 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:22 pm | 5/78 |

```

LINE # SOURCE TEXT
481 tagptr->pll_divisor = 256 - save_k + 1; /* normal case */
482 tagptr->clock_select = 1; /* select div by 2k */
483 }
484 return(SUCCESS);
485 }
486
487 /*
488  * int pac_stack_pams(lane_no)
489  * INPUT: lane_no = lane number to stack
490  * OUTPUT: return code = SUCCESS or FAILURE
491  * DESCRIPTION: Reads the PAC COUNT register to see
492  * how many PAMS are stacked onto the PAC. If there
493  * are none, the function returns FAILURE. If there
494  * is at least one PAM, the PAM PRESENT registers
495  * and the PAM ID PROMS are read. If the PAMS are
496  * strapped correctly, the PAC CONFIGURATION register
497  * is written, the PAC INFO structure is filled,
498  * and the function returns SUCCESS. If the PAM
499  * strapping is incorrect the function returns FAILURE.
500  */
501 int
502 pac_stack_pams(lane_no)
503 int lane_no;
504 {
505     if(pac[lane_no].exists != TRUE)
506     {
507         (void)lm_error("PAC in lane %c does not exist.\n", 'A' + (char)lane_no);
508         return(FAILURE);
509     }
510     if(pac_get_sum_pams(lane_no) != SUCCESS)
511     {
512         (void)lm_error("Could not determine number of PAMS in lane %c.\n",
513             'A' + (char)lane_no);
514         return(FAILURE);
515     }
516     if(pac_idprom_test(lane_no) != SUCCESS)
517     {
518         (void)lm_error("Bad ID Prom(s) in PAM(s) in lane %c.\n", 'A' +
519             (char)lane_no);
520         return(FAILURE);
521     }
522     if(pac_get_pam_types(lane_no) != SUCCESS)
523     {
524         (void)lm_error("Could not get PAM types in lane %c.\n", 'A' +
525             (char)lane_no);
526         return(FAILURE);
527     }
528     if(pac_configure_pac(lane_no) != SUCCESS)
529     {
530         (void)lm_error("Could not configure PAC in lane %c.\n", 'A' +
531             (char)lane_no);
532         return(FAILURE);
533     }
534     return(SUCCESS);
535 }
536
537 /*
538  * int pac_get_sum_pams(lane_no)
539  * INPUT: lane_no = lane number
540  * OUTPUT: return code = SUCCESS or FAILURE
541  * DESCRIPTION: Reads the PAC COUNT register to see
542  * how many PAMS are stacked onto the PAC. If there
543  * are none, the function returns FAILURE. If there
544  * is at least one PAM, the PAM PRESENT registers
545  * are read. If the PAMS are strapped correctly,
546  * the PAC INFO structure is filled, and the function
547  * returns SUCCESS. If the PAM strapping is incorrect
548  * the function returns FAILURE.
549  */
550 int
551 pac_get_sum_pams(lane_no)
552 int lane_no;
553 {
554     int pamno;
555     if(pac[lane_no].exists != TRUE)
556     {
557         (void)lm_error("PAC in lane %c does not exist.\n", 'A' + (char)lane_no);
558         return(FAILURE);
559     }
560     switch(pacptr[lane_no]->pam_count)
561     {
562         case ZERO_PAMS:
563             (void)lm_error("There are no PAMS on the PAC in lane %c.\n",
564                 'A' + (char)lane_no);
565             pac[lane_no].sum_pams = 0;
566             return(FAILURE);
567         case ONE_PAM:
568             pac[lane_no].sum_pams = 1;
569             break;
570         case TWO_PAMS:
571             pac[lane_no].sum_pams = 2;
572             break;
573         case THREE_PAMS:
574             pac[lane_no].sum_pams = 3;
575             break;
576         case FOUR_PAMS:
577             pac[lane_no].sum_pams = 4;
578             break;
579         default:
580             (void)lm_error("PAM Detect register in lane %c returns invalid code.\n",
581                 'A' + (char)lane_no);
582             pac[lane_no].sum_pams = 0;
583             return(FAILURE);
584     }
585     for(pamno = 0; pamno < pac[lane_no].sum_pams; pamno++)
586     {
587         if(pacptr[lane_no]->pam_register[pamno].present == PAM_NOT_PRESENT)
588         {
589             (void)lm_error("PAM %d in lane %c is strapped incorrectly.\n", pamno,
590                 'A' + (char)lane_no);
591             pac[lane_no].sum_pams = 0;
592             return(FAILURE);
593         }
594     }
595 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:22 pm | 6/79 |

```

LINE #          SOURCE TEXT
601      }
602      }
603      return(SUCCESS);
604      }
605
606      /*
607      int pac_get_pam_size(lane_no, pam_no)
608      *
609      * INPUT: lane_no = lane number
610      *       pam_no = Pattern memory board number
611      * OUTPUT: Number of patterns in the board
612      * DESCRIPTION: Reads the PAM ID from to determine the
613      *              number of patterns.
614      */
615      int
616      pac_get_pam_size(lane_no, pam_no)
617      int lane_no;
618      int pam_no;
619      {
620      ID FROM_PAM id_prom_table;
621      ID FROM_PAM *pam_id_info = id_prom_table;
622
623      (void)diag_get_id_info((int)((lane_no * LANE_SIZE) + LANE_A_PAM_ID_PROM +
624      (pam_no * PAM_ID_SPACE)), (char *)pam_id_info);
625
626      return(pam_id_info->patterns << 10);
627      }
628
629      /*
630      int pac_get_pam_types(lane_no)
631      *
632      * INPUT: lane_no = lane number
633      * OUTPUT: return code = SUCCESS or FAILURE
634      * DESCRIPTION: Checks the PAM ID Proms and if they
635      *              have proper checksums, reads the PAM ID in each.
636      *              If the PAMs are strapped correctly, the PAC INFO
637      *              structure is filled, and the function returns SUCCESS.
638      *              If the PAM strapping is incorrect the function returns
639      *              FAILURE.
640      */
641      int
642      pac_get_pam_types(lane_no)
643      int lane_no;
644      {
645      int pam_no;
646      register u_long current;
647      register u_long patterns;
648
649      pac[lane_no].num_blocks = 0;
650      pac[lane_no].num_patterns = 0;
651
652      if(pac[lane_no].exists != TRUE)
653      {
654      (void)lm_error("PAC in lane %c does not exist.\n", 'A' + (char)lane_no);
655      return(FAILURE);
656      }
657
658      current = PATTERNS_IN_2M;
659      for(pam_no = 0; pam_no < pac[lane_no].num_pams; pam_no++)
660      {
661      if((patterns = pac_get_pam_size(lane_no, pam_no)) > current)
662      {
663      (void)lm_error("PAMs in lane %c are strapped out of order.\n",
664      'A' + (char)lane_no);
665      pac[lane_no].num_patterns = 0;
666      return(FAILURE);
667      }
668      else
669      {
670      pac[lane_no].p_m_size[pam_no] = current - patterns;
671      pac[lane_no].num_patterns += patterns;
672      }
673      }
674
675      pac[lane_no].num_blocks = pac[lane_no].num_patterns / BLOCK_SIZE;
676      return(SUCCESS);
677      }
678
679      /*
680      int pac_configure_pac(lane_no)
681      *
682      * INPUT: lane_no = lane number of PAC to configure
683      * OUTPUT: return code = SUCCESS or FAILURE
684      * DESCRIPTION: Uses values stored in the PAC information
685      *              table (global) to write the PAC CONFIGURATION register.
686      */
687      int
688      pac_configure_pac(lane_no)
689      int lane_no;
690      {
691      if(pac[lane_no].exists != TRUE)
692      {
693      (void)lm_error("PAC in lane %c does not exist.\n", 'A' + (char)lane_no);
694      return(FAILURE);
695      }
696
697      switch(pac[lane_no].num_pams)
698      {
699      case 1:
700      switch(pac[lane_no].num_blocks)
701      {
702      case BLKS_IN_128K:
703      pacptr[lane_no]->configuration = 60;
704      break;
705      case BLKS_IN_512K:
706      pacptr[lane_no]->configuration = 61;
707      break;
708      case BLKS_IN_2M:
709      pacptr[lane_no]->configuration = 62;
710      break;
711      default:
712      (void)lm_error("Number of blocks in information table invalid.\n");
713      break;
714      }
715      break;
716      case 2:
717      case 3:
718      case 4:
719      pacptr[lane_no]->configuration = (pac[lane_no].num_blocks /
720      BLKS_IN_128K) - 1;
721      break;
722      default:
723      (void)lm_error("Number of PAMs in information table invalid.\n");
724      return(FAILURE);
725      }
726      }

```


Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM
diags/pac_util.cDATE 5/23/89
TIME 4:41:22 pmPAGE #
7/80

LINE # SOURCE TEXT

```

721 Pac_set_parity(lane_no, SET_ODD_PARITY);
722 return(SUCCESS);
723 }
724
725 /*
726  * int pac_fill_check(first_pattern, patterns)
727  *
728  * INPUT: first_pattern = first pattern number to fill (0 counting)
729  *        patterns = number of patterns to fill
730  * OUTPUT: return code = SUCCESS or FAILURE
731  * DESCRIPTION: Checks bounds on first pattern and patterns.
732  * Used by pac_fill_xx functions. The function returns
733  * SUCCESS if the patterns to be filled
734  * are within the pattern memory and FAILURE if any
735  * portion of the fill space is outside the physical
736  * pattern memory address space.
737  */
738 int
739 pac_fill_check(first_pattern, patterns)
740 int first_pattern;
741 int patterns;
742 {
743     if((first_pattern < 0) ||
744         (first_pattern > (pac(current_lane).num_patterns - 1)))
745     {
746         (void)lm_error("First pattern number passed to fill routine \
747 outside of pattern memory.\n");
748         return(FAILURE);
749     }
750     if((patterns < 1) || ((first_pattern + patterns) >
751         pac(current_lane).num_patterns))
752     {
753         (void)lm_error("Number of patterns passed to fill routine \
754 outside of pattern memory.\n");
755         return(FAILURE);
756     }
757     return(SUCCESS);
758 }
759
760 /*
761  * int pac_fill_random(first_pattern, patterns, fill_mask, seed)
762  *
763  * INPUT: first_pattern = first pattern number to fill
764  *        patterns = number of patterns to fill
765  *        fill_mask = control word mask
766  *        seed = random number seed
767  * OUTPUT: return code = SUCCESS or FAILURE
768  * DESCRIPTION: Fills pattern memory with pseudorandom
769  * data starting at the pattern number specified by 'first_pattern'.
770  * The total number of patterns is specified by 'patterns'.
771  * The function returns SUCCESS if the operation is
772  * successful and FAILURE if any portion of the fill
773  * space is outside the physical pattern memory address
774  * space.
775  */
776 int
777 pac_fill_random(first_pattern, patterns, fill_mask, seed)
778 int first_pattern;
779 int patterns;
780 register u_long fill_mask;
781 u_long seed;
782 {
783     register u_long *memptr;
784     register u_long i;
785     register u_long random_number;
786     u_long pattern_offset;
787     u_long bank_offset;
788     int bank;
789
790     if(pac_fill_check(first_pattern, patterns) != SUCCESS)
791         return(FAILURE);
792
793     pattern_offset = pac(current_lane).lane_offset + 4*first_pattern;
794     random_number = seed;
795
796     (void)lm_message("Filling to patterns with random data", patterns);
797     for(bank = 0; bank_offset = 0; bank < 3; ++bank)
798     {
799         (void)lm_message(" ");
800         memptr = (u_long *) (pattern_offset + bank_offset);
801         i = patterns / 4;
802         if((bank == 2) && (fill_mask != "01"))
803         {
804             do
805             {
806                 random_number = Pac_get_random(random_number);
807                 *memptr++ = random_number & fill_mask;
808                 random_number = Pac_get_random(random_number);
809                 *memptr++ = random_number & fill_mask;
810                 random_number = Pac_get_random(random_number);
811                 *memptr++ = random_number & fill_mask;
812                 random_number = Pac_get_random(random_number);
813                 *memptr++ = random_number & fill_mask;
814             } while (--i);
815             for(i=0; i < (patterns % 4); ++i)
816             {
817                 random_number = Pac_get_random(random_number);
818                 *memptr++ = random_number & fill_mask;
819             }
820         }
821         else
822         {
823             do
824             {
825                 random_number = Pac_get_random(random_number);
826                 *memptr++ = random_number;
827                 random_number = Pac_get_random(random_number);
828                 *memptr++ = random_number;
829                 random_number = Pac_get_random(random_number);
830                 *memptr++ = random_number;
831                 random_number = Pac_get_random(random_number);
832                 *memptr++ = random_number;
833             } while (--i);
834             for(i=0; i < (patterns % 4); ++i)
835             {
836                 random_number = Pac_get_random(random_number);
837                 *memptr++ = random_number;
838             }
839         }
840         bank_offset += PAT_MEM_BANK;

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_util.c

DATE 5/23/89

PAGE #

TIME 4:41:22 pm

8/81

```

LINE # SOURCE TEXT
841 }
842 (void)lm_message("done.\n");
843 return(SUCCESS);
844
845
846 /*
847  * int pac_read_random(first_pattern, patterns, fill_mask, seed)
848  *
849  * INPUT:  first_pattern = first pattern number to read
850  *         patterns = number of patterns to read
851  *         fill_mask = control word mask
852  *         seed = random number seed
853  *
854  * OUTPUT: return code = SUCCESS or FAILURE
855  *
856  * DESCRIPTION: Reads pattern memory and compares with pseudorandom
857  * data starting at the pattern number specified by 'first_pattern'.
858  * The total number of patterns is specified by 'patterns'.
859  * The function returns SUCCESS if all patterns are identical.
860  * The function returns FAILURE if any of the patterns do not
861  * compare. The failed address and the good and bad data words
862  * are output as an error message.
863  */
864 int
865 pac_read_random(first_pattern, patterns, fill_mask, seed)
866 {
867     register int first_pattern;
868     register int patterns;
869     register u_long fill_mask;
870     register u_long seed;
871     register u_long *memptr;
872     register u_long i;
873     register u_long temp;
874     register u_long random_number;
875     u_long pattern_offset;
876     u_long bank_offset;
877     int bank;
878     int returncode = SUCCESS;
879
880     if(pac_fill_check(first_pattern, patterns) != SUCCESS)
881         return(FAILURE);
882     pattern_offset = pac(current_lane).lane_offset + 4 * first_pattern;
883     random_number = seed;
884
885     (void)lm_message("Reading rd random data patterns", patterns);
886     for(bank = 0; bank_offset = 0; bank < 3; ++bank)
887     {
888         (void)lm_message("-");
889         memptr = (u_long *) (pattern_offset + bank_offset);
890         if((bank == 2) && (fill_mask != 0))
891         {
892             for(i=0; i < patterns; ++i)
893             {
894                 random_number = Pac_get_random(random_number);
895                 if((temp = *memptr++) != (random_number & fill_mask))
896                 {
897                     if(lm_error("Random read test. Address = %08x.\n",
898                         Expected = %08x, Actual = %08x.\n", memptr - 1, random_number & fill_mask,
899                         temp) != SUCCESS)
900                         return(FAILURE);
901                     else
902                         returncode = FAILURE;
903                 }
904             }
905         }
906         else
907         {
908             for(i=0; i < patterns; ++i)
909             {
910                 random_number = Pac_get_random(random_number);
911                 if((temp = *memptr++) != random_number)
912                 {
913                     if(lm_error("Random read test. Address = %08x.\n",
914                         Expected = %08x, Actual = %08x.\n", memptr - 1, random_number, temp) != SUCCESS)
915                         return(FAILURE);
916                     else
917                         returncode = FAILURE;
918                 }
919             }
920         }
921         bank_offset += PAT_MEM_BANK;
922     }
923     (void)lm_message("done.\n");
924     return(returncode);
925 }
926
927 /*
928  * int pac_fill_counting(first_pattern, patterns, fill_mask)
929  *
930  * INPUT:  first_pattern = first pattern
931  *         patterns = number of patterns to fill
932  *         fill_mask = control word mask
933  *
934  * OUTPUT: return code = SUCCESS or FAILURE
935  *
936  * DESCRIPTION: Fills pattern memory with 'counting' data
937  * starting at the pattern number specified by 'first_pattern'.
938  * The total number of patterns is specified by 'patterns'.
939  * The function returns SUCCESS if the operation is successful
940  * and FAILURE if any portion of the fill space is outside
941  * the physical pattern memory address space.
942  */
943 int
944 pac_fill_counting(first_pattern, patterns, fill_mask)
945 {
946     register int first_pattern;
947     register int patterns;
948     register u_long fill_mask;
949     register u_long i;
950     register u_long location;
951
952     if(pac_fill_check(first_pattern, patterns) != SUCCESS)
953         return(FAILURE);
954     location = pac(current_lane).lane_offset + 4 * first_pattern;
955     (void)lm_message("Filling rd patterns with counting data...", patterns);
956     for(i=0; i < patterns; ++i)
957     {
958         /* Write count to Banks 0 and 1 */
959         Write_long(location + BANK_0, i);
960         Write_long(location + BANK_1, i);
961         /* Write count to Bank 2. Masking off control bits */
962         Write_long(location + BANK_2, i & fill_mask);
963         location += 4; /* Increment location to next long word address */
964     }
965     (void)lm_message("done.\n");
966 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pac_util.c

DATE 5/23/89

PAGE #

TIME 4:41:22 pm

9/82

```

LINE # SOURCE TEXT
961 return(SUCCESS);
962 }
963
964
965 /* int pac_fill_load0(first_pattern, patterns, fill_mask)
966 *
967 * INPUT: first_pattern = first pattern
968 *         patterns = number of patterns to fill
969 *         fill_mask = control word mask
970 * OUTPUT: return code = SUCCESS or FAILURE
971 * DESCRIPTION: Fills pattern memory with alternating ones
972 * and zeros data starting at the pattern number specified by
973 * 'first_pattern'. The total number of patterns is specified by
974 * 'patterns'. The function returns SUCCESS if the operation
975 * is successful and FAILURE if any portion of the fill
976 * space is outside the physical pattern memory address space.
977 */
978
979 int
980 pac_fill_load0(first_pattern, patterns, fill_mask)
981 int first_pattern;
982 register int patterns;
983 register u_long fill_mask;
984 {
985     register u_long i;
986     register u_long value;
987     register u_long location;
988
989     if(pac_fill_check(first_pattern, patterns) != SUCCESS)
990         return(FAILURE);
991     location = pac[current_lane].lane_offset + 4*first_pattern;
992     (void)lm_message("Filling 0d patterns with alternating ones and \
993     zeros data...", patterns);
994     for(i=0; i < patterns; i++)
995     {
996         if((i & 1) == 0) /* If pattern number is even */
997             value = 0; /* value is all zeros */
998         else
999             value = ~0; /* value is all ones */
1000         /* Write value to Banks 0 and 1 */
1001         Write_long(location + BANK_0, value);
1002         Write_long(location + BANK_1, value);
1003         /* Write zeros to BANK 2, masking off control bits */
1004         Write_long(location + BANK_2, value & fill_mask);
1005         location += 4; /* Increment location to next long word address */
1006     }
1007     (void)lm_message("Done.\n");
1008     return(SUCCESS);
1009 }
1010
1011 /* int pac_fill_walking(first_pattern, patterns, fill_mask, data)
1012 *
1013 * INPUT: first_pattern = block address of first pattern
1014 *         patterns = number of patterns to fill
1015 *         fill_mask = control word mask
1016 *         data = pattern data (0 or 1)
1017 * OUTPUT: return code = SUCCESS or FAILURE
1018 * DESCRIPTION: Fills pattern memory with walking ones or
1019 * zeros data starting at the block number specified by
1020 * 'first_pattern'. The total number of patterns is specified by
1021 * 'patterns'. The function returns SUCCESS if the operation
1022 * is successful and FAILURE if any portion of the fill
1023 * space is outside the physical pattern memory address space.
1024 */
1025
1026 int
1027 pac_fill_walking(first_pattern, patterns, fill_mask, data)
1028 int first_pattern;
1029 register int patterns;
1030 u_long fill_mask;
1031 int data;
1032 {
1033     register u_long i;
1034     u_long same;
1035     u_long shift[32];
1036     register u_long location;
1037
1038     if(pac_fill_check(first_pattern, patterns) != SUCCESS)
1039         return(FAILURE);
1040     if(data == 0) /* Fill array with walking zeros */
1041     {
1042         same = 0;
1043         shift[0] = ~1;
1044         for(i=1; i < 32; ++i)
1045             shift[i] = (shift[i-1] << 1) | 0x01;
1046     }
1047     else /* Fill array with walking ones */
1048     {
1049         same = 0;
1050         shift[0] = 1;
1051         for(i=1; i < 32; ++i)
1052             shift[i] = (shift[i-1] << 1);
1053     }
1054     location = pac[current_lane].lane_offset + 4*first_pattern;
1055     (void)lm_message("Filling 0d patterns with walking %s data...",
1056     patterns, (data == 0) ? "zeros" : "ones");
1057     for(i=0; i < patterns; i++)
1058     {
1059         switch((i/32)%3)
1060         {
1061             case 0:
1062                 Write_long(location + BANK_0, shift[(i/32)]);
1063                 Write_long(location + BANK_1, same);
1064                 Write_long(location + BANK_2, same & fill_mask);
1065                 break;
1066             case 1:
1067                 Write_long(location + BANK_0, same);
1068                 Write_long(location + BANK_1, shift[(i/32)]);
1069                 Write_long(location + BANK_2, same & fill_mask);
1070                 break;
1071             case 2:
1072                 Write_long(location + BANK_0, same);
1073                 Write_long(location + BANK_1, same);
1074                 Write_long(location + BANK_2, shift[(i/32)] & fill_mask);
1075                 break;
1076             default:
1077                 return(FAILURE);
1078                 break;
1079         }
1080         location += 4; /* Increment location to next long word address */
1081     }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_util.c

DATE 5/23/89
TIME 4:41:22 pm

PAGE #
10/83

```

1081 (void)lm_message("done.\n");
1082 return(SUCCESS);
1083 }
1084
1085 /*
1086  * int location_test(address, first_bit, num_bits)
1087  *
1088  * INPUT: address = long word address
1089  *        first_bit = first valid bit in register (0-31)
1090  *        num_bits = number of valid bits in register (1-32)
1091  * OUTPUT: returns SUCCESS or FAILURE
1092  * DESCRIPTION: Performs complete register (or memory location)
1093  * test on location pointed to by address. The first and last
1094  * bits in the register determine which bits the function checks.
1095  */
1096 int
1097 location_test(address, first_bit, num_bits)
1098 u_long address;
1099 int first_bit;
1100 int num_bits;
1101 {
1102     u_long i;
1103     int j;
1104     u_long mask;
1105     u_long temp;
1106     int returncode = SUCCESS;
1107
1108     mask = (~01 << first_bit) & ~(~01 << (first_bit + num_bits));
1109
1110     /* Perform walking ones test */
1111     for(i = 1 << first_bit, j = 0; j < num_bits; j++, i <<= 1)
1112     {
1113         Write_long(address, i); /* Write pattern to location */
1114         if((temp = ((Read_long(address)) & mask)) != i)
1115         {
1116             returncode = FAILURE;
1117             if(lm_error("PAC location test. Address = %08x.\n",
1118 \expected = %08x.\nActual = %08x.\n", address, i, temp) != SUCCESS)
1119                 return(FAILURE);
1120         }
1121     }
1122
1123     /* Perform walking zeros test */
1124     for(i = (~01 << first_bit) & mask, j = 0; j < num_bits; j++,
1125 i = ((i << 1) | 0x01) & mask)
1126     {
1127         Write_long(address, i); /* Write pattern to location */
1128         if((temp = ((Read_long(address)) & mask)) != i)
1129         {
1130             returncode = FAILURE;
1131             if(lm_error("PAC location test. Address = %08x.\n",
1132 \expected = %08x.\nActual = %08x.\n", address, i, temp) != SUCCESS)
1133                 return(FAILURE);
1134         }
1135     }
1136     return(returncode);
1137 }
1138
1139 /*
1140  * int build_pattern_control(first_pattern, patterns, mode)
1141  *
1142  * INPUT: first_pattern = first pattern number to link
1143  *        patterns = number of patterns to link
1144  *        mode = STOP MODE or LOOP MODE
1145  * OUTPUT: return code = SUCCESS or FAILURE
1146  * DESCRIPTION: Links blocks of pattern memory with
1147  * link table. Puts Branch Always instructions in the
1148  * proper locations. In STOP MODE, a stop instruction is
1149  * placed at the end of the pattern sequence. In LOOP MODE,
1150  * a Branch Always instruction is placed at the end of the
1151  * pattern sequence and the link table is loaded so that
1152  * the last branch is to the initial pattern block. The
1153  * function returns SUCCESS or FAILURE.
1154  */
1155 int
1156 build_pattern_control(first_pattern, patterns, mode)
1157 int first_pattern;
1158 register int patterns;
1159 int mode;
1160 {
1161     register int i;
1162     register int j;
1163     register int max_index;
1164     register int total_branches;
1165     register u_long *memptr;
1166     u_long *linkptr;
1167     u_long pat_mem;
1168     u_long link_table;
1169     int first_block;
1170
1171     if((first_pattern < 0) ||
1172 (first_pattern > (pac(current_lane).num_patterns - 1)))
1173     {
1174         (void)lm_error("First pattern number passed to fill routine \
1175 outside of pattern memory.\n");
1176         return(FAILURE);
1177     }
1178     if((patterns < 1) || (patterns > pac(current_lane).num_patterns))
1179     {
1180         (void)lm_error("Number of patterns passed to fill routine \
1181 outside of pattern memory.\n");
1182         return(FAILURE);
1183     }
1184     if((first_pattern & BLOCK_SIZE) != 0)
1185     {
1186         (void)lm_error("First pattern in seq not on block boundary.\n");
1187         return(FAILURE);
1188     }
1189     if((mode == LOOP_MODE) && ((patterns & BLOCK_SIZE) != 0))
1190     {
1191         (void)lm_error("Can only loop in block increments.\n");
1192         return(FAILURE);
1193     }
1194
1195     /* Compute first block number and total number of branches */
1196     first_block = first_pattern/BLOCK_SIZE;
1197     pacptr(current_lane)->branch_address = first_block;
1198     total_branches = (patterns - 1)/BLOCK_SIZE;
1199
1200     /* Compute first address of pattern memory and link table */

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_util.c

DATE 5/23/89
TIME 4:41:22 pm

PAGE #
11/84

```

1201 pat_mem = pac(current_lane).lane_offset + BANK_2;
1202 memptr = (u_long *) (pat_mem + 4 * first_patterns);
1203 link_table = pac(current_lane).lane_offset + LINK_OFFSET;
1204 linkptr = (u_long *) (link_table + 4 * first_block);
1205 /* See if patterns sequence wraps around pattern memory */
1206 if ((first_patterns + patterns) <= pac(current_lane).sum_patterns) /* No */
1207 {
1208     /* Put NOP instructions (Don't branch or stop) in pattern sequence */
1209     for (i=0; i < patterns; i++)
1210     {
1211         *(memptr++) = NOP_MASK;
1212     }
1213
1214     /* Fill link table and place branch always instructions */
1215     memptr = (u_long *) (pat_mem + 4 * (first_patterns + BLOCK_SIZE -
1216     BRANCH_LATENCY));
1217     for (i = first_block + 1, j = 0; j < total_branches; i++, j++)
1218     {
1219         *(linkptr++) = i; /* Connect blocks in link table */
1220         *memptr |= BRANCH_ALWAYS; /* Place branch always instruction */
1221         memptr += BLOCK_SIZE; /* Go to next branch location */
1222     }
1223
1224     /* patterns sequence wraps around to beginning of pattern memory */
1225     /* Put NOP instructions (Don't branch or stop) in pattern sequence */
1226     max_index = pac(current_lane).sum_patterns - first_patterns;
1227     for (i=0; i < max_index; i++)
1228     {
1229         *(memptr++) = NOP_MASK;
1230     }
1231     memptr = (u_long *) (pat_mem);
1232     max_index = patterns - max_index;
1233     for (i=0; i < max_index; i++)
1234     {
1235         *(memptr++) = NOP_MASK;
1236     }
1237
1238     /* Fill link table and place branch always instructions */
1239     memptr = (u_long *) (pat_mem + 4 * (first_patterns + BLOCK_SIZE -
1240     BRANCH_LATENCY));
1241     max_index = pac(current_lane).sum_blocks - first_block;
1242     for (i = first_block + 1, j = 0; j < max_index; i++, j++)
1243     {
1244         *(linkptr++) = i; /* Connect blocks in link table */
1245         *memptr |= BRANCH_ALWAYS; /* Place branch always instruction */
1246         memptr += BLOCK_SIZE; /* Go to next branch location */
1247     }
1248     *(--linkptr) = 0; /* Point back to first block */
1249     linkptr = (u_long *) (link_table);
1250     memptr = (u_long *) (pat_mem + 4 * (BLOCK_SIZE - BRANCH_LATENCY));
1251     max_index = total_branches - max_index;
1252     for (i = 0; i < max_index; i++)
1253     {
1254         *(linkptr++) = i + 1; /* Connect blocks in link table */
1255         *memptr |= BRANCH_ALWAYS; /* Place branch always instruction */
1256         memptr += BLOCK_SIZE; /* Go to next branch location */
1257     }
1258
1259     linkptr = first_block; /* Go back to first block */
1260
1261     /* Place stop or branch instruction, depending on mode */
1262     if (mode == LOOP_MODE)
1263     {
1264         *memptr |= BRANCH_ALWAYS; /* Place branch always instruction */
1265     }
1266     else /* must be STOP_MODE */
1267     {
1268         memptr = (u_long *) (pat_mem + 4 * ((first_patterns + patterns -
1269     STOP_LATENCY) & pac(current_lane).sum_patterns));
1270         *memptr |= STOP; /* Place stop instruction */
1271     }
1272     return (SUCCESS);
1273 }
1274
1275 void pac_clear_link()
1276 {
1277     /*
1278     * INPUT: none
1279     * OUTPUT: none
1280     * DESCRIPTION: Clears link table in current lane.
1281     */
1282 }
1283
1284 void pac_clear_pam()
1285 {
1286     register u_long *memptr;
1287     register u_long i;
1288
1289     memptr = (u_long *) (pac(current_lane).lane_offset + LINK_OFFSET);
1290
1291     i = (LINK_SIZE >> 2) / 8;
1292     do
1293     {
1294         *memptr++ = 0;
1295         *memptr++ = 0;
1296         *memptr++ = 0;
1297         *memptr++ = 0;
1298         *memptr++ = 0;
1299         *memptr++ = 0;
1300         *memptr++ = 0;
1301         *memptr++ = 0;
1302     } while (--i);
1303 }
1304
1305 int pac_clear_pam(lane_no, pam_no)
1306 {
1307     /*
1308     * INPUT: lane_no = lane number of PAC/PAMs
1309     * pam_no = PAM number
1310     * OUTPUT: returns code = SUCCESS or FAILURE
1311     * DESCRIPTION: Clears pattern memory on specified PAM.
1312     */
1313     int
1314     pac_clear_pam(lane_no, pam_no)
1315     int lane_no;
1316     int pam_no;
1317     {
1318         register u_long *memptr;
1319         register u_long i;
1320         register u_long total_patterns;

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_util.c

DATE 5/23/89
TIME 4:41:22 pm

PAGE #
12/85

```

1321 int bank,
1322 u_long bank_offset,
1323
1324 if(pac(lane_no).exists != TRUE)
1325 {
1326     (void)lm_error("PAC in lane %d does not exist.\n", 'A' + (char)lane_no);
1327     return(FAILURE);
1328 }
1329 if ((total_patterns = pac(lane_no).pam_size(pam_no)) == 0)
1330 {
1331     return SUCCESS; /* nothing to clear */
1332 }
1333 (void)lm_message("Clearing PAM %d (bank) in lane %d", pam_no,
1334 total_patterns >> 10, 'A' + (char)lane_no);
1335 for(bank = 0, bank_offset = 0; bank < 3; ++bank)
1336 {
1337     lm_message("."); /* walking period before each bank */
1338     pamptr = (u_long - 1)(Pam_base_address(lane_no, pam_no) + bank_offset);
1339     i = total_patterns / 8;
1340     do
1341     {
1342         *pamptr++ = 0;
1343         *pamptr++ = 0;
1344         *pamptr++ = 0;
1345         *pamptr++ = 0;
1346         *pamptr++ = 0;
1347         *pamptr++ = 0;
1348         *pamptr++ = 0;
1349         *pamptr++ = 0;
1350     } while(--i);
1351     bank_offset += PAT_NUM_BANK;
1352 }
1353 (void)lm_message("Done.\n");
1354 return(SUCCESS);
1355 }
1356
1357 /*
1358 int pac_clear_pam_mem(lane_no)
1359 *
1360 * INPUT: lane_no = lane number of PAC to clear
1361 * OUTPUT: return code = SUCCESS or FAILURE
1362 * DESCRIPTION: Clears all of pattern memory.
1363 */
1364 int
1365 pac_clear_pam_mem(lane_no)
1366 int lane_no,
1367 {
1368     int pam_no;
1369     for(pam_no = 0; pam_no < pac(lane_no).num_pams; pam_no++)
1370     {
1371         if(pac_clear_pam(lane_no, pam_no) != SUCCESS)
1372         {
1373             (void)lm_error("Unable to clear pattern memory in lane %d.\n",
1374 'A' + (char)lane_no);
1375             return(FAILURE);
1376         }
1377     }
1378     return(SUCCESS);
1379 }
1380
1381 /*
1382 int pac_compute_playtime(count, clock_period)
1383 *
1384 * INPUT: count = length of pattern play in physical clocks
1385 *        clock_period = clock period in ns
1386 * OUTPUT: play time in ns
1387 * DESCRIPTION: Computes duration of pattern play in milliseconds.
1388 */
1389 int
1390 pac_compute_playtime(count, clock_period)
1391 int count,
1392 int clock_period;
1393 {
1394     return(((int)((double)count * (double)clock_period) /
1395 1000000.0));
1396 }
1397
1398 #define MIN_SAMPLE_PULSE 500 /* Min sample pulse width = 500 ns */
1399
1400 /*
1401 int pac_compute_sample_width(clock_period)
1402 *
1403 * INPUT: clock_period = clock period in ns
1404 * OUTPUT: sample width in clock cycles
1405 * DESCRIPTION: Computes sample width for pattern play.
1406 */
1407 int
1408 pac_compute_sample_width(clock_period)
1409 int clock_period;
1410 {
1411     int sample_width;
1412     sample_width = (u_long)MIN_SAMPLE_PULSE / clock_period;
1413     return((sample_width < 1) ? 1 : sample_width);
1414 }
1415
1416 /*
1417 void pac_probe_all_pacs()
1418 *
1419 * INPUT: none
1420 * OUTPUT: none
1421 * DESCRIPTION: Probes all PACs in the modeler and
1422 *             fills in the .exists PAC information attributes.
1423 */
1424 void
1425 pac_probe_all_pacs()
1426 {
1427     register int lane;
1428     for(lane = 0; lane < NUMBER_OF_LANES; lane++)
1429     {
1430         pac(lane).exists = (probe_pac(lane) == SUCCESS)? TRUE : FALSE;
1431     }
1432 }
1433
1434 /*
1435 void pac_configure_all_pacs()
1436 *
1437 * INPUT: none
1438 * OUTPUT: none
1439 * DESCRIPTION: Configures all PACs in the modeler.
1440 */

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_util.c

DATE 5/23/89
TIME 4:41:22 pm

PAGE #
13/86

```

1441 void
1442 pac_configure_all_pacs()
1443 {
1444     int lane_no;
1445
1446     pac_info_init(); /* Initialize PAC information structure */
1447     pac_probe_all_pacs();
1448     for(lane_no = 0; lane_no < NUMBER_OF_LANES; lane_no++)
1449     {
1450         if (pac(lane_no).exists == TRUE)
1451         {
1452             if(pac_stack_pacs(lane_no) == SUCCESS)
1453             {
1454                 if(pac_clear_pat_mem(lane_no) == SUCCESS)
1455                 {
1456                     configured_lanes |= Lane_code(lane_no);
1457                 }
1458             }
1459         }
1460     }
1461
1462     /*
1463     *      int pac_insert_parity_error(pattern, word_no)
1464     *
1465     *      INPUT:  pattern = pattern number to insert parity error
1466     *              word_no = word number within pattern number
1467     *      OUTPUT: returns code = SUCCESS or FAILURE
1468     *      DESCRIPTION: Inserts a parity error in pattern memory
1469     *                  at the location specified by the pattern number and the
1470     *                  word number. The word number is encoded as follows
1471     *                  0 = control word
1472     *                  1 = pattern bits 0-15
1473     *                  2 = pattern bits 16-31
1474     *                  3 = pattern bits 32-47
1475     *                  4 = pattern bits 48-63
1476     *                  5 = pattern bits 64-79
1477     */
1478     int
1479     pac_insert_parity_error(pattern, word_no)
1480     int pattern;
1481     int word_no;
1482     {
1483         u_long bank;
1484         u_long word;
1485         u_long address;
1486         u_long contents;
1487
1488         /* Check bounds on pattern */
1489         if((pattern < 0) || (pattern > (pac(current_lane).num_patterns - 1)))
1490         {
1491             (void)ls_error("Pattern number out of bounds.\n");
1492             return(FAILURE);
1493         }
1494         /* decode word number */
1495         switch(word_no)
1496         {
1497             case 0:
1498                 bank = BANK_2;
1499                 word = 2;
1500                 break;
1501             case 1:
1502                 bank = BANK_2;
1503                 word = 0;
1504                 break;
1505             case 2:
1506                 bank = BANK_1;
1507                 word = 2;
1508                 break;
1509             case 3:
1510                 bank = BANK_1;
1511                 word = 0;
1512                 break;
1513             case 4:
1514                 bank = BANK_0;
1515                 word = 2;
1516                 break;
1517             case 5:
1518                 bank = BANK_0;
1519                 word = 0;
1520                 break;
1521             default:
1522                 (void)ls_error("Word number out of bounds.\n");
1523                 return(FAILURE);
1524                 break;
1525         }
1526         /* Compute address */
1527         address = pac(current_lane).lane_offset + bank + (4 * pattern) + word;
1528         contents = Read_word(address);
1529
1530         /* change parity to opposite of current state */
1531         if(word == 0)
1532             pacptr(current_lane)->high_word_parity =
1533             !pacptr(current_lane)->high_word_parity;
1534         else
1535             pacptr(current_lane)->low_word_parity =
1536             !pacptr(current_lane)->low_word_parity;
1537         Write_word(address, contents);
1538
1539         /* change parity back to original state */
1540         if(word == 0)
1541             pacptr(current_lane)->high_word_parity =
1542             !pacptr(current_lane)->high_word_parity;
1543         else
1544             pacptr(current_lane)->low_word_parity =
1545             !pacptr(current_lane)->low_word_parity;
1546         return(SUCCESS);
1547     }
1548
1549     /*
1550     *      int pac_remove_parity_error(pattern, word_no)
1551     *
1552     *      INPUT:  pattern = pattern number with parity error
1553     *              word_no = word number within pattern number
1554     *      OUTPUT: returns code = SUCCESS or FAILURE
1555     *      DESCRIPTION: Removes a parity error in pattern memory
1556     *                  at the location specified by the pattern number and the
1557     *                  word number. The word number is encoded as follows
1558     *                  0 = control word
1559     *                  1 = pattern bits 0-15
1560     */

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pac_util.c | DATE 5/23/89 | PAGE # 14/87 |
|--|---|------------------------------------|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 1561 | * | 2 = patterns bits 16-31 | | |
| 1562 | * | 3 = patterns bits 32-47 | | |
| 1563 | * | 4 = patterns bits 48-63 | | |
| 1564 | * | 5 = patterns bits 64-79 | | |
| 1565 | */ | | | |
| 1566 | int | | | |
| 1567 | pac_remove_parity_error(patterns, word_no) | | | |
| 1568 | int patterns; | | | |
| 1569 | int word_no; | | | |
| 1570 | { | | | |
| 1571 | u_long bank; | | | |
| 1572 | u_long word; | | | |
| 1573 | u_long address; | | | |
| 1574 | u_long contents; | | | |
| 1575 | /* | | | |
| 1576 | /* Check bounds on patterns */ | | | |
| 1577 | if((patterns < 0) (patterns > (pac[current_lane].num_patterns - 1))) | | | |
| 1578 | { | | | |
| 1579 | (void)lm_error("Pattern number out of bounds.\n"); | | | |
| 1580 | return(FAILURE); | | | |
| 1581 | } | | | |
| 1582 | /* decode word number */ | | | |
| 1583 | switch(word_no) | | | |
| 1584 | { | | | |
| 1585 | case 0: | | | |
| 1586 | bank = BANK_2; | | | |
| 1587 | word = 2; | | | |
| 1588 | break; | | | |
| 1589 | case 1: | | | |
| 1590 | bank = BANK_2; | | | |
| 1591 | word = 0; | | | |
| 1592 | break; | | | |
| 1593 | case 2: | | | |
| 1594 | bank = BANK_1; | | | |
| 1595 | word = 2; | | | |
| 1596 | break; | | | |
| 1597 | case 3: | | | |
| 1598 | bank = BANK_1; | | | |
| 1599 | word = 0; | | | |
| 1600 | break; | | | |
| 1601 | case 4: | | | |
| 1602 | bank = BANK_0; | | | |
| 1603 | word = 2; | | | |
| 1604 | break; | | | |
| 1605 | case 5: | | | |
| 1606 | bank = BANK_0; | | | |
| 1607 | word = 0; | | | |
| 1608 | break; | | | |
| 1609 | default: | | | |
| 1610 | (void)lm_error("Word number out of bounds.\n"); | | | |
| 1611 | return(FAILURE); | | | |
| 1612 | break; | | | |
| 1613 | } | | | |
| 1614 | /* Compute address */ | | | |
| 1615 | address = pac[current_lane].lane_offset + bank + (4 * patterns) + word; | | | |
| 1616 | | | | |
| 1617 | contents = Read_word(address); | | | |
| 1618 | Write_word(address, contents); | | | |
| 1619 | Clear_pac_errors(current_lane); | | | |
| 1620 | return(SUCCESS); | | | |
| 1621 | } | | | |
| 1622 | } | | | |
| 1623 | /* | | | |
| 1624 | int pac_get_pam_number(patterns_no) | | | |
| 1625 | /* | | | |
| 1626 | INPUT: patterns_no = pattern number | | | |
| 1627 | OUTPUT: Returns board number containing pattern number | | | |
| 1628 | DESCRIPTION: Determines which Pattern Memory board | | | |
| 1629 | contains the specified pattern number. The function | | | |
| 1630 | returns -1 if the pattern number is outside of pattern | | | |
| 1631 | memory. | | | |
| 1632 | */ | | | |
| 1633 | int | | | |
| 1634 | pac_get_pam_number(patterns_no) | | | |
| 1635 | register int patterns_no; | | | |
| 1636 | { | | | |
| 1637 | register int total_patterns; | | | |
| 1638 | register int pam_no; | | | |
| 1639 | register int total_pams = pac[current_lane].num_pams; | | | |
| 1640 | | | | |
| 1641 | if((patterns_no < 0) (patterns_no > (pac[current_lane].num_patterns - 1))) | | | |
| 1642 | return(-1); | | | |
| 1643 | | | | |
| 1644 | for(total_patterns = 0, pam_no = 0; pam_no < total_pams; ++pam_no) | | | |
| 1645 | { | | | |
| 1646 | total_patterns += pac[current_lane].pam_size[pam_no]; | | | |
| 1647 | if(patterns_no < total_patterns) | | | |
| 1648 | return(pam_no); | | | |
| 1649 | } | | | |
| 1650 | return(-1); | | | |
| 1651 | } | | | |
| 1652 | /* | | | |
| 1653 | int pac_get_first_pattern_no(lane_no, pam_no) | | | |
| 1654 | /* | | | |
| 1655 | INPUT: lane_no = lane number | | | |
| 1656 | pam_no = pattern memory board number | | | |
| 1657 | OUTPUT: Returns first pattern number in specified pattern memory board | | | |
| 1658 | DESCRIPTION: Determines the first pattern number (zeros counting) | | | |
| 1659 | in the specified Pattern Memory board. | | | |
| 1660 | */ | | | |
| 1661 | int | | | |
| 1662 | pac_get_first_pattern_no(lane_no, pam_no) | | | |
| 1663 | register int lane_no; | | | |
| 1664 | register int pam_no; | | | |
| 1665 | { | | | |
| 1666 | register int first_pattern = 0; | | | |
| 1667 | register int i; | | | |
| 1668 | | | | |
| 1669 | for(i = 0; i < pam_no; ++i) | | | |
| 1670 | first_pattern += pac[lane_no].pam_size[i]; | | | |
| 1671 | return(first_pattern); | | | |
| 1672 | } | | | |
| 1673 | /* | | | |
| 1674 | int pac_replay() | | | |
| 1675 | /* | | | |
| 1676 | INPUT: none | | | |
| 1677 | OUTPUT: return code = SUCCESS or FAILURE | | | |
| 1678 | DESCRIPTION: Performs a pattern play from pattern | | | |
| 1679 | number entered by user. Assumes play has just been | | | |
| 1680 | | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pac_util.c

DATE 5/23/89
TIME 4:41:22 pm

PAGE #
15/88

| LINE # | SOURCE TEXT |
|--------|---|
| 1681 | /* performed to get default pattern number. |
| 1682 | */ |
| 1683 | int |
| 1684 | pac_replay() |
| 1685 | { |
| 1686 | int first_patterns; |
| 1687 | int last_block; |
| 1688 | int input_good; |
| 1689 | |
| 1690 | /* Find out where last pattern play left off */ |
| 1691 | last_block = pacptr(current_lane) -> branch_address; |
| 1692 | /* Look up first block number in link table and compute pattern */ |
| 1693 | first_patterns = (Read_long(pac[current_lane].lane_offset + LINK_OFFSET + |
| 1694 | (last_block << 2) + 0x7fff) + BLOCK_SIZE; |
| 1695 | do |
| 1696 | { |
| 1697 | input_good = TRUE; |
| 1698 | diag_get_ubox((u_long *) &first_patterns, "first pattern number (hex)", 01, |
| 1699 | (u_long)(pac[current_lane].num_patterns - BLOCK_SIZE)); |
| 1700 | if((first_patterns % BLOCK_SIZE) != 0) |
| 1701 | { |
| 1702 | (void)la_message("First pattern must be multiple of 256 (100 hex).\n"); |
| 1703 | input_good = FALSE; |
| 1704 | } |
| 1705 | } while(input_good != TRUE); |
| 1706 | |
| 1707 | pac_set_first_block(Lane_code(current_lane), first_patterns / BLOCK_SIZE); |
| 1708 | if(diag_play()) != SUCCESS) |
| 1709 | { |
| 1710 | (void)la_error("Pattern play failed trying to play from pattern %X.\n", |
| 1711 | first_patterns); |
| 1712 | return(FAILURE); |
| 1713 | } |
| 1714 | return(SUCCESS); |
| 1715 | } |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/parity.c | DATE 5/23/89 | PAGE # 1/89 |
|--|---|----------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCCS_ID: parity.c rev 3.1, 4/24/89 at 07:49:32 */ | | | |
| 2 | /* | | | |
| 3 | ** Parity error handler | | | |
| 4 | ** | | | |
| 5 | */ | | | |
| 6 | #include "common.h" | | | |
| 7 | #include "cpu.h" | | | |
| 8 | #include "mod_err.h" | | | |
| 9 | #include "svram.h" | | | |
| 10 | #include "lm_rd_wr.h" | | | |
| 11 | extern u_short lm_svram_access(); | | | |
| 12 | extern void output_routine(), reset_cpu(); | | | |
| 13 | parity_error() | | | |
| 14 | { | | | |
| 15 | register cpu_control_reg_struct *control_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG; | | | |
| 16 | register cpu_par_err_reg *parity_reg = (cpu_par_err_reg *) CPU_PAR_ERR_REG, parity; | | | |
| 17 | u_long error; | | | |
| 18 | char buffer[100]; | | | |
| 19 | parity = *parity_reg; | | | |
| 20 | if (control_reg->not_parity_intr == 0) | | | |
| 21 | { | | | |
| 22 | sprintf(buffer, "Parity error addr = %08x\n", | | | |
| 23 | parity_reg->error_addr + 4); | | | |
| 24 | output_routine(buffer); | | | |
| 25 | sprintf(buffer, "%020 id vme id LANCE id\n", | | | |
| 26 | parity_reg->not_68020_master, | | | |
| 27 | parity_reg->not_VME_master, | | | |
| 28 | parity_reg->not_LANCE_master); | | | |
| 29 | output_routine(buffer); | | | |
| 30 | sprintf(buffer, "hi id um id lm id lo id\n", | | | |
| 31 | parity_reg->not_error_hi, | | | |
| 32 | parity_reg->not_error_um, | | | |
| 33 | parity_reg->not_error_lm, | | | |
| 34 | parity_reg->not_error_lo); | | | |
| 35 | output_routine(buffer); | | | |
| 36 | (void)lm_svram_access((char *) parity_reg, PARITY_ERR, sizeof(PARITY_ERR), MEMORY_WRITE, &error); | | | |
| 37 | control_reg->parity_force= 0; | | | |
| 38 | reset_cpu(PARITY_ERROR); | | | |
| 39 | } | | | |
| 40 | } | | | |
| 41 | u_long network_timeout; | | | |

| | | | |
|---|------|------------|----------------|
| S | DATE | 5/23/89 | PAGE # 1/90 |
| | TIME | 4:41:23 pm | |

[illegible]

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pat_bus.c | DATE 5/23/89 | PAGE # 2/91 |
|--|---|-----------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 121 | if(!m_error("Play failed during pattern bit test.\n") != SUCCESS) | | | |
| 122 | return(FAILURE); | | | |
| 123 | { | | | |
| 124 | if(pel_xor_pattern_bits(end_pattern, bad_bits) | | | |
| 125 | != SUCCESS) | | | |
| 126 | { | | | |
| 127 | returncode = FAILURE; | | | |
| 128 | if(!m_error("Pattern bits did not match.\n") != SUCCESS) | | | |
| 129 | return(FAILURE); | | | |
| 130 | } | | | |
| 131 | set_pattern_bit(end_pattern, pattern_bit, 0); | | | |
| 132 | } | | | |
| 133 | if(returncode == FAILURE) | | | |
| 134 | { | | | |
| 135 | /* Print out all bad bits */ | | | |
| 136 | (void)m_message("Bad pattern bits: "); | | | |
| 137 | for(magic_chip = 0; magic_chip < 5; ++magic_chip) | | | |
| 138 | (void)m_message("%04X", bad_bits[magic_chip]); | | | |
| 139 | (void)m_message("\n"); | | | |
| 140 | } | | | |
| 141 | return(returncode); | | | |
| 142 | } | | | |
| 143 | } | | | |
| 144 | } | | | |
| 145 | int | | | |
| 146 | pel_check_for_parity_errors() | | | |
| 147 | { | | | |
| 148 | register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel)); | | | |
| 149 | register int chip; | | | |
| 150 | register int returncode = SUCCESS; | | | |
| 151 | { | | | |
| 152 | for(chip = 0; chip < 5; ++chip) | | | |
| 153 | { | | | |
| 154 | if(pel->magic_chip[chip].m.parity_out != 0) | | | |
| 155 | { | | | |
| 156 | returncode = FAILURE; | | | |
| 157 | if(!m_error("Magic chip ad detected a parity error.\n", chip) != SUCCESS) | | | |
| 158 | return(FAILURE); | | | |
| 159 | } | | | |
| 160 | } | | | |
| 161 | return(returncode); | | | |
| 162 | } | | | |
| 163 | } | | | |
| 164 | } | | | |
| 165 | #define PEL_0_CTL_0 0x7f1 | | | |
| 166 | int | | | |
| 167 | pattern_bus_test(first_pattern, num_patterns) | | | |
| 168 | int first_pattern, | | | |
| 169 | int num_patterns, | | | |
| 170 | { | | | |
| 171 | register long *memptr; | | | |
| 172 | register int load_patterns = num_patterns - 1; | | | |
| 173 | register int pel_no = current_pel; | | | |
| 174 | int returncode = SUCCESS; | | | |
| 175 | { | | | |
| 176 | /* Initialize the current lane for pattern play */ | | | |
| 177 | if(pel_lane_init() != SUCCESS) | | | |
| 178 | { | | | |
| 179 | (void)m_error("Unable to initialize lane for pattern bus test.\n"); | | | |
| 180 | return(FAILURE); | | | |
| 181 | } | | | |
| 182 | /* Clear any MAGIC errors and make sure the PEL can't assert a PLAY error */ | | | |
| 183 | /* on the backplane */ | | | |
| 184 | pel_disable_bp_error(current_lane, current_pel); | | | |
| 185 | /* Fill pat-ctrl memory with random data */ | | | |
| 186 | if(pac_fill_random(first_pattern, num_patterns, PEL_0_CTL_0, SEED) != SUCCESS) | | | |
| 187 | { | | | |
| 188 | (void)m_error("Unable to fill memory with random data in pattern bus test.\n"); | | | |
| 189 | return(FAILURE); | | | |
| 190 | } | | | |
| 191 | /* Now OR in the current PEL and the PEL control bits 010 in the first */ | | | |
| 192 | /* location and 100 in the rest */ | | | |
| 193 | memptr = (long *) pattern_ctrl_addr(current_lane, 2, first_pattern); | | | |
| 194 | *memptr++ = pel_no (0x2 << 4); | | | |
| 195 | do | | | |
| 196 | { | | | |
| 197 | *memptr++ = pel_no (0x4 << 4); | | | |
| 198 | } | | | |
| 199 | while(--load_patterns); | | | |
| 200 | if(build_patterns_control(first_pattern, num_patterns, STOP_MODE) != SUCCESS) | | | |
| 201 | { | | | |
| 202 | (void)m_error("Unable to build pattern control in pattern bus test.\n"); | | | |
| 203 | return(FAILURE); | | | |
| 204 | } | | | |
| 205 | if(diag_play() != SUCCESS) | | | |
| 206 | { | | | |
| 207 | (void)m_error("Pattern play failed during pattern bus test.\n"); | | | |
| 208 | return(FAILURE); | | | |
| 209 | } | | | |
| 210 | /* Make sure there weren't any lane errors */ | | | |
| 211 | if(pac_check_errors(current_lane, m_error) != SUCCESS) | | | |
| 212 | { | | | |
| 213 | Clear_pac_errors(current_lane); | | | |
| 214 | (void)m_error("Pattern play caused a Pattern Controller error.\n"); | | | |
| 215 | returncode = FAILURE; | | | |
| 216 | } | | | |
| 217 | if(pel_check_for_parity_errors() != SUCCESS) | | | |
| 218 | { | | | |
| 219 | pel_disable_bp_error(current_lane, current_pel); | | | |
| 220 | (void)m_error("Pattern play caused a Pin Electronics error.\n"); | | | |
| 221 | returncode = FAILURE; | | | |
| 222 | } | | | |
| 223 | return(returncode); | | | |
| 224 | } | | | |
| 225 | } | | | |
| 226 | } | | | |
| 227 | int | | | |
| 228 | pel_patterns_bus_test() | | | |
| 229 | { | | | |
| 230 | if(diag_clear_errors() != SUCCESS) | | | |
| 231 | return(FAILURE); | | | |
| 232 | } | | | |
| 233 | return(pattern_bus_test(0, PATTERNS_IN_128K)); | | | |
| 234 | } | | | |
| 235 | } | | | |
| 236 | } | | | |
| 237 | int | | | |
| 238 | pel_pattern_bits_test() | | | |
| 239 | { | | | |
| 240 | { | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pat_bus.c

DATE 5/23/89

PAGE #

TIME 4:41:23 pm

3/92

```

LINE # SOURCE TEXT
241 if(diag_clear_errors() != SUCCESS)
242 return(FAILURE);
243
244 return(pattern_bits_test(0));
245 }
246
247
248 int
249 pel_check_parity(pattern, error_chip)
250 int pattern;
251 int error_chip;
252 {
253     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
254     PAT_WORD pattern_word, *patptr = &pattern_word;
255     long diff;
256     long memory;
257     long sum_bits;
258     long parity;
259     int chip;
260     int i;
261     int returncode = SUCCESS;
262
263     /* Read the pattern with the parity error */
264     (void)pac_read_pattern(pattern, patptr);
265
266     /* Verify that only the proper magic chip detected a parity error */
267     /* and that the data compares in each chip */
268     for(chip = 0; chip < 5; ++chip)
269     {
270         if(chip == error_chip) /* Should have detected error */
271         {
272             if(pel->magic_chip[chip].m.parity_out != 1)
273             {
274                 returncode = FAILURE;
275                 if(!error("Magic chip %d did not detect error.\n", chip) != SUCCESS)
276                     return(FAILURE);
277             }
278         }
279         else /* Should not have detected error */
280         {
281             if(pel->magic_chip[chip].m.parity_out != 0)
282             {
283                 returncode = FAILURE;
284                 if(!error("Magic chip %d detected error.\n", chip) != SUCCESS)
285                     return(FAILURE);
286             }
287         }
288         /* Read the word stored in memory and compute odd parity */
289         memory = patptr->pattern_data[4 - chip];
290         for(sum_bits = 0, i = 0; i < 16; ++i)
291             sum_bits += (memory >> i) & 1;
292         parity = sum_bits & 1;
293
294         if(pel->magic_chip[chip].m.parity_in != ((chip == error_chip) ?
295             ('Parity') & 1 : parity))
296         {
297             returncode = FAILURE;
298             if(!error("Magic chip %d latched incorrect parity sense.\n", chip)
299                 != SUCCESS)
300                 return(FAILURE);
301         }
302         if((diff = pel->magic_chip[chip].m.error_data - memory) != 0)
303         {
304             returncode = FAILURE;
305             if(!error("Magic chip %d latched the following bad bits: %04x.\n",
306                 chip, diff) != SUCCESS)
307                 return(FAILURE);
308         }
309     }
310     return(returncode);
311 }
312
313 #define PEL_ERROR_DELAY 6
314
315 /*
316  * int pel_parity_error_test()
317  *
318  * INPUT: none
319  * OUTPUT: returncode = SUCCESS or FAILURE
320  * DESCRIPTION:
321  */
322 int
323 pel_parity_error_test()
324 {
325     register int pel_no = current_pel;
326     register long *memptr;
327     int pattern;
328     int timeout;
329     int branch_address[3];
330     int block_offset[3];
331     int actual_branch;
332     int actual_block;
333     int magic_chip;
334     int i;
335     int returncode = SUCCESS;
336
337     if(diag_clear_errors() != SUCCESS)
338         return(FAILURE);
339
340     /* Initialize the current lane for pattern play */
341     if(pel_lane_init() != SUCCESS)
342     {
343         (void)lm_error("Unable to initialize lane for pattern bits test.\n");
344         return(FAILURE);
345     }
346
347     if(dab_type == NO_DAB) /* Make sure we don't get a play error */
348         pel_disable_bp_error(current_lane, current_pel);
349     else
350         pel_dab_init(PUBLIC_DAB);
351
352     /* Fill a block of pattern memory with random data, starting with block 1 */
353     if(pac_fill_random(BLOCK_SIZE, BLOCK_SIZE, PEL_0_CTL_0, SEED) != SUCCESS)
354     {
355         (void)lm_error("Could not fill block number one with random data.\n");
356         return(FAILURE);
357     }
358     /* Now OR in the current PEL and the PEL control bits 010 in the first */
359     /* location and 100 in the rest */
360     memptr = (long *)Pattern_to_address(current_lane, 2, BLOCK_SIZE);

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pat_bus.c

DATE 5/23/89
TIME 4:41:23 pm

PAGE #
4/93

| LINE # | SOURCE TEXT |
|--------|--|
| 361 | /*memptr++ = pel_no (dab << 4); |
| 362 | pattern = BLOCK_SIZE - 1; |
| 363 | do |
| 364 | { |
| 365 | /*memptr++ = pel_no (dab << 4); |
| 366 | } |
| 367 | while(--pattern); |
| 368 | if(build_pattern_control(BLOCK_SIZE, BLOCK_SIZE, STOP_MODE) != SUCCESS) |
| 369 | { |
| 370 | (void)lm_error("Could not build pattern control in block number one.\n"); |
| 371 | return(FAILURE); |
| 372 | } |
| 373 | /* Prepare for pattern play and compute a conservative timeout */ |
| 374 | if((timeout = pec_pre_play()) == 0) |
| 375 | { |
| 376 | (void)lm_error("Pattern play preparation failed.\n"); |
| 377 | return(FAILURE); |
| 378 | } |
| 379 | for(pattern = BLOCK_SIZE + 1; pattern < ((2 * BLOCK_SIZE) - 1); ++pattern) |
| 380 | { |
| 381 | if(dab_type != NO_DAB) |
| 382 | { |
| 383 | for(i = 0; i < 3; ++i) |
| 384 | { |
| 385 | if(pec_predict_offsets(pattern, PEL_ERROR_DELAY + 1, &branch_address[i], |
| 386 | &block_offset[i]) != SUCCESS) |
| 387 | { |
| 388 | (void)lm_error("Unable to predict parity error offsets for pattern \ |
| 389 | number %d, error delay = %d.\n", pattern, PEL_ERROR_DELAY + 1); |
| 390 | return(FAILURE); |
| 391 | } |
| 392 | } |
| 393 | } |
| 394 | for(magic_chip = 0; magic_chip < 5; ++magic_chip) |
| 395 | { |
| 396 | if(pec_insert_parity_error(pattern, magic_chip + 1) != SUCCESS) |
| 397 | { |
| 398 | (void)lm_error("Could not insert parity error for magic chip %d.\n", |
| 399 | magic_chip); |
| 400 | return(FAILURE); |
| 401 | } |
| 402 | pecptr[current_lane]-->branch_address = 1; |
| 403 | if(pec_play(timeout) != SUCCESS) |
| 404 | { |
| 405 | (void)lm_error("Pattern play failed.\n"); |
| 406 | return(FAILURE); |
| 407 | } |
| 408 | /* If there's a DAB, see if pattern play generated a backplane error */ |
| 409 | if((dab_type != NO_DAB) && ((Get_bp_error() & lane_code(current_lane)) |
| 410 | == 0)) |
| 411 | { |
| 412 | (void)lm_error("Pattern play did not generate error (should have).\n"); |
| 413 | return(FAILURE); |
| 414 | } |
| 415 | if(pec_check_parity(pattern, magic_chip) != SUCCESS) |
| 416 | { |
| 417 | returncode = FAILURE; |
| 418 | if(lm_error("Magic chip parity error verification failed.\n") |
| 419 | != SUCCESS) |
| 420 | return(FAILURE); |
| 421 | } |
| 422 | /* Check PAC offsets if there is a DAB */ |
| 423 | if(dab_type != NO_DAB) |
| 424 | { |
| 425 | actual_branch = pecptr[current_lane]-->branch_address; |
| 426 | if((actual_branch != branch_addresses[0]) && |
| 427 | (actual_branch != branch_addresses[1]) && |
| 428 | (actual_branch != branch_addresses[2])) |
| 429 | { |
| 430 | returncode = FAILURE; |
| 431 | if(lm_error("Branch address for pattern %d is incorrect.\n\ |
| 432 | tExpected %d, %d or %d. Actual %d.\n", pattern, branch_addresses[0], |
| 433 | branch_addresses[1], branch_addresses[2], actual_branch) != SUCCESS) |
| 434 | return(FAILURE); |
| 435 | } |
| 436 | actual_block = pecptr[current_lane]-->block_offset; |
| 437 | if((actual_block != block_offset[0]) && |
| 438 | (actual_block != block_offset[1]) && |
| 439 | (actual_block != block_offset[2])) |
| 440 | { |
| 441 | returncode = FAILURE; |
| 442 | if(lm_error("Block offset for pattern %d is incorrect.\n\ |
| 443 | tExpected %d, %d or %d. Actual %d.\n", pattern, block_offset[0], |
| 444 | block_offset[1], block_offset[2], actual_block) != SUCCESS) |
| 445 | return(FAILURE); |
| 446 | } |
| 447 | } |
| 448 | if(pec_remove_parity_error(pattern, magic_chip + 1) != SUCCESS) |
| 449 | { |
| 450 | (void)lm_error("Could not remove parity error for magic chip %d.\n", |
| 451 | magic_chip); |
| 452 | return(FAILURE); |
| 453 | } |
| 454 | /* Get rid of any PEL errors */ |
| 455 | if(dab_type == NO_DAB) |
| 456 | pel_disable_bp_error(current_lane, current_pel); |
| 457 | else |
| 458 | pel_clear_pel_errors(); |
| 459 | } |
| 460 | } |
| 461 | return(returncode); |
| 462 | } |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel.c

DATE 5/23/89
TIME 4:41:24 pm

PAGE #
1/94

```

1  /* SCCS ID: pel.c rev 3.1, 4/24/89 at 07:49:40 */
2
3  .....
4  pel_menu.c
5  .....
6  Main Menu
7  used in PEL diagnostics
8  .....
9
10 #include <math.h>
11 #include "common.h"
12 #include "lm_diags.h"
13 #include "vrtx.h"
14 #include "tmg.h"
15 #include "mod_def.h"
16 #include "modeler_extn.h"
17 #include "magic.h"
18 #include "pel.h"
19 #include "id.h"
20 #include "eeprom.h"
21
22 int current_pel = 0;
23 int dab_type = NO_DAB;
24
25 #define PEL_TEST_INDEX 0
26 #define PEL_DIAG_DAB_TEST_INDEX 1
27
28 pel_menu(parent_menu, info)
29 lm_diag_menu *parent_menu;
30 char *info;
31 {
32     char buffer[132];
33     static char test_buffer[132];
34     static char diag_dab_test_buffer[132];
35     register char *buf;
36
37     int pel_test_menu();
38     int pel_diag_dab_test_menu();
39     int pel_test_utilities();
40     int pel_play_utilities();
41     int pel_patterns_utilities();
42     extern int lm_acceptance;
43
44     static lm_diag_menu_item menu_list[] =
45     {
46         {
47             "1",
48             test_buffer,
49             pel_test_menu,
50             lm_diag_another_menu,
51             lm_diag_null,
52             0
53         },
54         {
55             "2",
56             diag_dab_test_buffer,
57             pel_diag_dab_test_menu,
58             0,
59             lm_diag_null,
60             0
61         },
62         {
63             "3",
64             "PEL Test Utilities",
65             pel_test_utilities,
66             lm_diag_utility_menu,
67             lm_diag_null,
68             0
69         },
70         {
71             "4",
72             "PEL Pattern Utilities",
73             pel_patterns_utilities,
74             lm_diag_utility_menu,
75             lm_diag_null,
76             0
77         },
78         {
79             "5",
80             "PEL Play Utilities",
81             pel_play_utilities,
82             lm_diag_utility_menu,
83             lm_diag_null,
84             0
85         },
86     },
87     1,
88     static lm_diag_menu menu =
89     {
90         0,
91         sizeof(menu_list) / sizeof(lm_diag_menu_item),
92         0,
93         menu_list
94     },
95     .....
96
97     current_pel = (long)(info - '0');
98     if (Host) {
99         dab_type = DIAG_DAB;
100     } else {
101         dab_type = what_dab(current_lane, current_pel);
102     }
103
104     if (lm_acceptance && (dab_type != DIAG_DAB))
105         return SUCCESS;
106
107     menu_list[PEL_DIAG_DAB_TEST_INDEX].attributes
108     = lm_diag_another_menu | lm_diag_disable;
109     switch (dab_type) {
110         case NO_DAB:
111             buf = "NO";
112             break;
113         case DIAG_DAB:
114             buf = "DIAGNOSTIC";
115             menu_list[PEL_TEST_INDEX].attributes
116             |= lm_diag_acceptance;
117             menu_list[PEL_DIAG_DAB_TEST_INDEX].attributes
118             |= lm_diag_acceptance;
119             /* turn off the disable bit only if there is a PAC
120

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel.c

DATE 5/23/89
TIME 4:41:24 pm

PAGE #
2/95

| LINE # | SOURCE TEXT |
|--------|---|
| 121 | in the lane */ |
| 122 | if(probe_poc(current_lane) == SUCCESS) |
| 123 | { |
| 124 | menu_list[PEL_DIAG_DAB_TEST_INDEX].attributes |
| 125 | = LM_DIAG_disable; |
| 126 | } |
| 127 | break; |
| 128 | case UNKNOWN_DAB: |
| 129 | default: |
| 130 | buf = "NON-DIAGNOSTIC"; |
| 131 | break; |
| 132 | } |
| 133 | |
| 134 | sprintf(buf, "%s (%s DAB)", parent_menu-> |
| 135 | menu_items[parent_menu->current_selection].menu_text, buf); |
| 136 | menu.title = buf; |
| 137 | |
| 138 | sprintf(test_buffer, "Lane to PEL to Test Menu (%s DAB)", |
| 139 | current_lane + 'A', current_pel, buf); |
| 140 | |
| 141 | sprintf(diag_dab_test_buffer, |
| 142 | "Lane to PEL to Diagnostic Adapter Test Menu (%s DAB)", |
| 143 | current_lane + 'A', current_pel, buf); |
| 144 | |
| 145 | return lm_display_menus(&menu); |
| 146 | |
| 147 | |
| 148 | pel_test_menu(parent_menu) |
| 149 | LM_DIAG_MENU *parent_menu; |
| 150 | { |
| 151 | register long i; |
| 152 | register int pec_present; |
| 153 | register LM_DIAG_MENU_ITEM *m; |
| 154 | |
| 155 | int pel_idprom_checksum(); |
| 156 | int pel_wr_car(); |
| 157 | int pel_wr_magic_ctl(); |
| 158 | int pel_dac_adc(); |
| 159 | int pel_patterns_control(); |
| 160 | int pel_patterns_bits_test(); |
| 161 | int pel_patterns_bus_test(); |
| 162 | int pel_parity_error_test(); |
| 163 | int pel_edge(); |
| 164 | |
| 165 | static LM_DIAG_MENU_ITEM menu_list[] = |
| 166 | { |
| 167 | { |
| 168 | "1", |
| 169 | "Pis Electronics ID Prom Test", |
| 170 | pel_idprom_checksum, |
| 171 | LM_DIAG_diag_routine LM_DIAG_acceptance, |
| 172 | LM_DIAG_null, |
| 173 | (char *) (DIAG_DAB NO_DAB UNKNOWN_DAB) |
| 174 | }, |
| 175 | { |
| 176 | "2", |
| 177 | "Control/Status Register Test", |
| 178 | pel_wr_car, |
| 179 | LM_DIAG_diag_routine LM_DIAG_acceptance, |
| 180 | LM_DIAG_null, |
| 181 | (char *) (DIAG_DAB NO_DAB UNKNOWN_DAB) |
| 182 | }, |
| 183 | { |
| 184 | "3", |
| 185 | "DACs and ADCs Test", |
| 186 | pel_dac_adc, |
| 187 | LM_DIAG_diag_routine LM_DIAG_acceptance, |
| 188 | LM_DIAG_null, |
| 189 | (char *) (DIAG_DAB NO_DAB UNKNOWN_DAB) |
| 190 | }, |
| 191 | { |
| 192 | "4", |
| 193 | "Magic Chip Control Register Test", |
| 194 | pel_wr_magic_ctl, |
| 195 | LM_DIAG_diag_routine LM_DIAG_acceptance, |
| 196 | LM_DIAG_null, |
| 197 | (char *) (DIAG_DAB NO_DAB UNKNOWN_DAB NEED_PAC) |
| 198 | }, |
| 199 | { |
| 200 | "5", |
| 201 | "Patterns Control Test", |
| 202 | pel_patterns_control, |
| 203 | LM_DIAG_diag_routine LM_DIAG_acceptance, |
| 204 | LM_DIAG_null, |
| 205 | (char *) (DIAG_DAB NO_DAB UNKNOWN_DAB NEED_PAC) |
| 206 | }, |
| 207 | { |
| 208 | "6", |
| 209 | "Patterns Bit Test", |
| 210 | pel_patterns_bits_test, |
| 211 | LM_DIAG_diag_routine LM_DIAG_acceptance, |
| 212 | LM_DIAG_null, |
| 213 | (char *) (DIAG_DAB NO_DAB UNKNOWN_DAB NEED_PAC) |
| 214 | }, |
| 215 | { |
| 216 | "7", |
| 217 | "Patterns Bus Test", |
| 218 | pel_patterns_bus_test, |
| 219 | LM_DIAG_diag_routine LM_DIAG_acceptance, |
| 220 | LM_DIAG_null, |
| 221 | (char *) (DIAG_DAB NO_DAB UNKNOWN_DAB NEED_PAC) |
| 222 | }, |
| 223 | { |
| 224 | "8", |
| 225 | "Parity Error Detection Test", |
| 226 | pel_parity_error_test, |
| 227 | LM_DIAG_diag_routine LM_DIAG_acceptance, |
| 228 | LM_DIAG_null, |
| 229 | (char *) (DIAG_DAB NO_DAB UNKNOWN_DAB NEED_PAC) |
| 230 | }, |
| 231 | { |
| 232 | "9", |
| 233 | "Timing Edge Test", |
| 234 | pel_edge, |
| 235 | LM_DIAG_diag_routine LM_DIAG_acceptance, |
| 236 | LM_DIAG_null, |
| 237 | (char *) (DIAG_DAB NO_DAB UNKNOWN_DAB NEED_PAC) |
| 238 | }, |
| 239 | }; |
| 240 | } |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pel.c

DATE 5/23/89

PAGE #

TIME 4:41:24 pm

3/96

```

LINE # SOURCE TEXT
241 static LM_DIAG_MENU menu =
242 {
243     0,
244     sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
245     0,
246     menu_list
247 },
248
249 /*=====*/
250
251 if (Host) {
252     dab_type = DIAG_DAB,
253 } else {
254     dab_type = what_dab(current_lane, current_pel);
255     pac_present = probe_pac(current_lane);
256     for (m = menu_list, i = 0; i < menu.number_of_items; ++i, ++m) {
257         if (((long)(m->user_data) & dab_type) {
258             m->attributes |= LM_DIAG_disable;
259         } else {
260             m->attributes |= LM_DIAG_disable;
261             if (((long)(m->user_data) & NEED_PAC) != 0) &&
262                 (pac_present != SUCCESS))
263                 m->attributes |= LM_DIAG_disable;
264         }
265     }
266
267     menu.title = parent_menu->
268     menu_items[parent_menu->current_selection].menu_text,
269     return lm_display_menu(menu);
270 }
271
272 pel_diag_dab_test_menu(parent_menu)
273 LM_DIAG_MENU *parent_menu;
274 {
275     register long i;
276     register LM_DIAG_MENU_ITEM *m;
277
278     int pel_opens_and_shorts();
279     int pel_crc();
280     int pel_feedback_test();
281     int pel_short_sensor_test();
282     int pel_error_test();
283     int pel_keeplive_test();
284     int pel_driver_test();
285     int pel_comparator_test();
286
287     static LM_DIAG_MENU_ITEM menu_list[] =
288     {
289         {
290             "1",
291             "Diagnostic Adapter Opens/Shorts Test",
292             pel_opens_and_shorts,
293             LM_DIAG_diag_routine|LM_DIAG_acceptance,
294             LM_DIAG_null,
295             (char *) (DIAG_DAB)
296         },
297         {
298             "2",
299             "Diagnostic Adapter CRC Test",
300             pel_crc,
301             LM_DIAG_diag_routine|LM_DIAG_acceptance,
302             LM_DIAG_null,
303             (char *) (DIAG_DAB)
304         },
305         {
306             "3",
307             "Diagnostic Adapter Feedback Test",
308             pel_feedback_test,
309             LM_DIAG_diag_routine|LM_DIAG_acceptance,
310             LM_DIAG_null,
311             (char *) (DIAG_DAB)
312         },
313         {
314             "4",
315             "Diagnostic Adapter Short Sensor Test",
316             pel_short_sensor_test,
317             LM_DIAG_diag_routine|LM_DIAG_acceptance,
318             LM_DIAG_null,
319             (char *) (DIAG_DAB)
320         },
321         {
322             "5",
323             "Diagnostic Adapter Error Handling Test",
324             pel_error_test,
325             LM_DIAG_diag_routine|LM_DIAG_acceptance,
326             LM_DIAG_null,
327             (char *) (DIAG_DAB)
328         },
329         {
330             "6",
331             "Diagnostic Adapter Keelalive/Trigger Bit Test",
332             pel_keeplive_test,
333             LM_DIAG_diag_routine|LM_DIAG_acceptance,
334             LM_DIAG_null,
335             (char *) (DIAG_DAB)
336         },
337         {
338             "7",
339             "Diagnostic Adapter LM-1000 Driver Test",
340             pel_driver_test,
341             LM_DIAG_diag_routine|LM_DIAG_acceptance,
342             LM_DIAG_null,
343             (char *) (DIAG_DAB)
344         },
345         {
346             "8",
347             "Diagnostic Adapter LM-1000 Receiver Test",
348             pel_comparator_test,
349             LM_DIAG_diag_routine|LM_DIAG_acceptance,
350             LM_DIAG_null,
351             (char *) (DIAG_DAB)
352         },
353     },
354
355     /* ADD THESE TESTS
356     *
357     * 1. Address decoding (CPU access and pattern play)
358     * 2. Measure currents on drivers (soft/medium/hard)
359     * 3. timing measurement
360     * 4. complete comparator test (LM-100 receiver test)

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel.c

DATE 5/23/89
TIME 4:41:24 pm

PAGE #
4/97

```

LINE # SOURCE TEXT
361 //
362
363
364 static LM_DIAG_MENU menu =
365 {
366     0,
367     sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
368     0,
369     menu_list
370 },
371
372 /*****
373
374 if (Host) {
375     dab_type = DIAG_DAB;
376 } else {
377     dab_type = what_dab(current_lane, current_pel);
378     for (m = menu_list, i = 0; i < menu.number_of_items; ++i, ++m) {
379         if ((long)(m->user_data) & dab_type) {
380             m->attributes |= LM_DIAG_disable;
381         } else {
382             m->attributes |= LM_DIAG_disable;
383         }
384     }
385 }
386
387 menu.title = parent_menu->
388 menu.items[parent_menu->current_selection].menu_text;
389 return lm_display_menu(menu);
390 }
391
392 pel_test_utilities(parent_menu)
393 LM_DIAG_MENU *parent_menu;
394 {
395     int pel_type_patterns_string();
396     int display_magic_control();
397     int pel_set_pel_ctrl();
398     show_all_adc();
399     int loop_adc_read();
400     int loop_dac_toggle();
401     int display_patterns_memory();
402     int display_patterns_preamble();
403     int display_patterns_preamble_versosely();
404     int print_magic_status();
405     int display_pel_errors();
406
407     static LM_DIAG_MENU_ITEM menu_list[] =
408     {
409         {
410             "1",
411             "Display and set pel control register",
412             pel_set_pel_ctrl,
413             LM_DIAG_utility,
414             LM_DIAG_null,
415             0
416         },
417         {
418             "2",
419             "Display MAGIC control register",
420             display_magic_control,
421             LM_DIAG_utility,
422             LM_DIAG_null,
423             0
424         },
425         {
426             "3",
427             "Display MAGIC status register",
428             print_magic_status,
429             LM_DIAG_utility,
430             LM_DIAG_null,
431             0
432         },
433         {
434             "4",
435             "Display PEL errors",
436             display_pel_errors,
437             LM_DIAG_utility,
438             LM_DIAG_null,
439             0
440         },
441         {
442             "5",
443             "Show all ADC values",
444             show_all_adc,
445             LM_DIAG_utility,
446             LM_DIAG_null,
447             0
448         },
449         {
450             "6",
451             "Read ADC indefinitely",
452             loop_adc_read,
453             LM_DIAG_utility,
454             LM_DIAG_null,
455             0
456         },
457         {
458             "7",
459             "Toggle DAC outputs indefinitely",
460             loop_dac_toggle,
461             LM_DIAG_utility,
462             LM_DIAG_null,
463             0
464         },
465     },
466
467     static LM_DIAG_MENU menu =
468     {
469         "PEL UTILITIES",
470         sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
471         0,
472         menu_list
473     },
474
475     menu.title = parent_menu->
476     menu.items[parent_menu->current_selection].menu_text;
477     return lm_display_menu(menu);
478 }
479
480 pel_check_csr(csr, value)
481 register unsigned short *csr;

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel.c

DATE 5/23/89
TIME 4:41:24 pm

PAGE #
5/98

```

481 u_short value;
482 {
483     *car = value;
484     if ((*car & 0xff) != value) {
485         lm_message("Host 402x, Read 402x\n", value, (*car & 0xff));
486         return (-1);
487     } else {
488         return (0);
489     }
490 }
491
492 pel_set_pel_car()
493 {
494     long answer;
495     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
496
497     lm_message("Pel car = 404x\n", pel->car.reg);
498     lm_message("magic_errorL (%d)\n", pel->car.bit.magic_errorL);
499     lm_message("play_errorL (%d)\n", pel->car.bit.play_errorL);
500     lm_message("errorL (%d)\n", pel->car.bit.errorL);
501     lm_message("present (%d)\n", pel->car.bit.present);
502     lm_message("active (%d)\n", pel->car.bit.active);
503     lm_message("eeprom_out (%d)\n", pel->car.bit.eeprom_out);
504
505     answer = pel->car.bit.eeprom_sel;
506     diag_get_long(&answer, "eeprom_sel", 01, 11);
507     pel->car.bit.eeprom_sel = answer;
508
509     answer = pel->car.bit.eeprom_clk;
510     diag_get_long(&answer, "eeprom_clk", 01, 11);
511     pel->car.bit.eeprom_clk = answer;
512
513     answer = pel->car.bit.eeprom_in;
514     diag_get_long(&answer, "eeprom_in", 01, 11);
515     pel->car.bit.eeprom_in = answer;
516
517     answer = pel->car.bit.initialize;
518     diag_get_long(&answer, "initialize", 01, 11);
519     pel->car.bit.initialize = answer;
520
521     answer = pel->car.bit.magic_error_enableL;
522     diag_get_long(&answer, "magic_error_enableL", 01, 11);
523     pel->car.bit.magic_error_enableL = answer;
524
525     answer = pel->car.bit.private;
526     diag_get_long(&answer, "private", 01, 11);
527     pel->car.bit.private = answer;
528
529     answer = pel->car.bit.resetL;
530     diag_get_long(&answer, "resetL", 01, 11);
531     pel->car.bit.resetL = answer;
532
533     answer = pel->car.bit.in_use_led;
534     diag_get_long(&answer, "in_use_led", 01, 11);
535     pel->car.bit.in_use_led = answer;
536 }
537
538 pel_wr_car()
539 {
540     register PEL *pel
541         = (PEL *) (pel_addr(current_lane, current_pel));
542     register unsigned short *car = &(pel->car.reg);
543
544     register long errors = 0;
545
546     if (!Host) {
547         errors += pel_check_car(car, 0x00);
548         errors += pel_check_car(car, 0x01);
549         errors += pel_check_car(car, 0x02);
550         errors += pel_check_car(car, 0x04);
551         errors += pel_check_car(car, 0x08);
552         errors += pel_check_car(car, 0x10);
553         errors += pel_check_car(car, 0x20);
554         errors += pel_check_car(car, 0x40);
555         errors += pel_check_car(car, 0x80);
556         errors += pel_check_car(car, 0x00);
557     } else {
558         lm_message("Can't run this test on the host\n");
559     }
560
561     lm_message("Alone to / pel id / address tx / ", current_lane + 'A', current_pel, car);
562     lm_message("PEL size tx\n", sizeof(PEL));
563     pel_display_car(pel);
564
565     if (errors) {
566         return(FAILURE);
567     } else {
568         return(SUCCESS);
569     }
570 }
571
572 pel_display_car(pel)
573 {
574     register PEL *pel;
575
576     lm_message("PEL CSR(404X):\n", pel->car.reg);
577     lm_message("MAGICERROR = %d\n", pel->car.bit.magic_errorL);
578     lm_message("PLAYERROR = %d\n", pel->car.bit.play_errorL);
579     lm_message("ERROR = %d\n", pel->car.bit.errorL);
580     lm_message("PRESENT = %d\n", pel->car.bit.present);
581     lm_message("ACTIVE = %d\n", pel->car.bit.active);
582     lm_message("EEDOUT = %d\n", pel->car.bit.eeprom_out);
583     lm_message("EESSEL = %d\n", pel->car.bit.eeprom_sel);
584     lm_message("EECLK = %d\n", pel->car.bit.eeprom_clk);
585     lm_message("EEMIN = %d\n", pel->car.bit.eeprom_in);
586     lm_message("INITIALIZE = %d\n", pel->car.bit.initialize);
587     lm_message("MAGIC ERREN = %d\n", pel->car.bit.magic_error_enableL);
588     lm_message("PRIVATE = %d\n", pel->car.bit.private);
589     lm_message("RESET = %d\n", pel->car.bit.resetL);
590     lm_message("INUSE LED = %d\n", pel->car.bit.in_use_led);
591 }
592
593 pel_idprom_checksum()
594 {
595     register PEL *pel
596         = (PEL *) (pel_addr(current_lane, current_pel));
597     ID_PROM PEL id;
598
599     if (!Host) {
600         lm_message("Can't read pel id prom on host\n");
        return FAILURE;
    }

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel.c | DATE 5/23/89 | PAGE # 6/99 |
|--|--|---|-----------------|----------------|
| LINE # | | SOURCE TEXT | | |
| 601 | | } | | |
| 602 | | } | | |
| 603 | | pel_id_load(pel, tid); | | |
| 604 | | lm_message("\sPEL base address = %08x\n", pel); | | |
| 605 | | lm_message("Board type = %08x\n", id.generic.board_type); | | |
| 606 | | lm_message("Revision = %c\n", id.generic.revision); | | |
| 607 | | lm_message("ECO level = %08x\n", id.generic.eco_level); | | |
| 608 | | lm_message("Speed = %08x\n", id.speed); | | |
| 609 | | lm_message("Pattern width = %08x\n", id.width * ID_GEN_WIDTH_K); | | |
| 610 | | lm_message("Pattern memory = %08x\n", id.pattmem); | | |
| 611 | | lm_message("MAGIC fab date = %08x08x\n", id.magic_build.year, id.magic_build.week); | | |
| 612 | | lm_message("Checksum = %08x\n", id.checksum); | | |
| 613 | | if (id.generic.board_type != ID_BT_TILPEL) { | | |
| 614 | | (void)pel_error("ID From is not TILPEL\n"); | | |
| 615 | | return(FAILURE); | | |
| 616 | | } | | |
| 617 | | if (!pel_id_check(pel)) { | | |
| 618 | | return SUCCESS; | | |
| 619 | | } else { | | |
| 620 | | return FAILURE; | | |
| 621 | | } | | |
| 622 | | } | | |
| 623 | | } | | |
| 624 | | } | | |
| 625 | | } | | |
| 626 | | } | | |
| 627 | | what_dab(lane, pel_no) | | |
| 628 | | int lane; | | |
| 629 | | int pel_no; | | |
| 630 | | { | | |
| 631 | | register PEL *pel = (PEL *) (pel_addr(lane, pel_no)); | | |
| 632 | | DAB_EEPROM dab; | | |
| 633 | | if (Host) { | | |
| 634 | | lm_message("Can't probe on host\n"); | | |
| 635 | | return UNKNOWN_DAB; | | |
| 636 | | } | | |
| 637 | | pel->car.reg = 0; /* make sure ERM=0 (for Diagnostic DAB) */ | | |
| 638 | | if (!pel->car.bit.present) { | | |
| 639 | | dab_type = NO_DAB; | | |
| 640 | | return(NO_DAB); | | |
| 641 | | } else { | | |
| 642 | | if ((lm_read_eeprom(pel_no + (lane << 3), &dab) | | |
| 643 | | == SUCCESS) && !strcmp(dab.device_name, "DIAGNOSTIC_ADAPTER")) { | | |
| 644 | | dab_type = DIAG_DAB; | | |
| 645 | | return(DIAG_DAB); | | |
| 646 | | } else { | | |
| 647 | | dab_type = UNKNOWN_DAB; | | |
| 648 | | return(UNKNOWN_DAB); | | |
| 649 | | } | | |
| 650 | | } | | |
| 651 | | } | | |
| 652 | | } | | |
| 653 | | } | | |
| 654 | | } | | |
| 655 | | } | | |
| 656 | | } | | |
| 657 | | } | | |
| 658 | | loop_adc_read() | | |
| 659 | | { | | |
| 660 | | register PEL *pel | | |
| 661 | | = (PEL *) (pel_addr(current_lane, current_pel)); | | |
| 662 | | int dummy; | | |
| 663 | | flush_key_buf(); | | |
| 664 | | lm_message("Hit key to stop..."); | | |
| 665 | | while (!lm_check_key()) { | | |
| 666 | | dummy = pel->vwith_adc; | | |
| 667 | | lm_delay(1); | | |
| 668 | | dummy = pel->vwhth_adc; | | |
| 669 | | lm_delay(1); | | |
| 670 | | } | | |
| 671 | | return dummy; /* abort list up */ | | |
| 672 | | } | | |
| 673 | | } | | |
| 674 | | } | | |
| 675 | | } | | |
| 676 | | } | | |
| 677 | | loop_dac_toggle() | | |
| 678 | | { | | |
| 679 | | register PEL *pel | | |
| 680 | | = (PEL *) (pel_addr(current_lane, current_pel)); | | |
| 681 | | flush_key_buf(); | | |
| 682 | | lm_message("Hit key to stop..."); | | |
| 683 | | while (!lm_check_key()) { | | |
| 684 | | pel->vlogl_dac = 0; | | |
| 685 | | pel->vlogh_dac = 0; | | |
| 686 | | pel->vwith_dac = 0; | | |
| 687 | | pel->val_dac = 0; | | |
| 688 | | pel->vwhth_dac = 0; | | |
| 689 | | pel->vwh_dac = 0; | | |
| 690 | | lm_delay(1); | | |
| 691 | | pel->vlogl_dac = 0x0f; | | |
| 692 | | pel->vlogh_dac = 0x0f; | | |
| 693 | | pel->vwith_dac = 0x0f; | | |
| 694 | | pel->val_dac = 0x0f; | | |
| 695 | | pel->vwhth_dac = 0x0f; | | |
| 696 | | pel->vwh_dac = 0x0f; | | |
| 697 | | lm_delay(1); | | |
| 698 | | } | | |
| 699 | | } | | |
| 700 | | } | | |
| 701 | | } | | |
| 702 | | } | | |
| 703 | | } | | |
| 704 | | } | | |
| 705 | | display_eeprom_info() | | |
| 706 | | { | | |
| 707 | | DAB_EEPROM dab; | | |
| 708 | | char configbuf[256]; | | |
| 709 | | if (lm_read_eeprom(current_pel + (current_lane << 3), &dab) != SUCCESS) { | | |
| 710 | | (void)pel_error("Cannot read DAB EEPROM\n"); | | |
| 711 | | return(FAILURE); | | |
| 712 | | } | | |
| 713 | | construct_cfg_str(configbuf, dab.configuration); | | |
| 714 | | lm_message(" Device Adapter in Lane %c Slot %d:\n", | | |
| 715 | | current_lane + 'A', current_pel); | | |
| 716 | | lm_message(" signature: %20.4s ", dab.signature); | | |
| 717 | | lm_message(" insertion count: %u\n", dab.insertion_count); | | |
| 718 | | } | | |
| 719 | | } | | |
| 720 | | } | | |

[illegible]

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel.c | DATE 5/23/89 | PAGE # 8/101 |
|--|--|---|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 841 | | int pel_type_pattern_string(); | | |
| 842 | | int pel_set_timing(); | | |
| 843 | | int pel_set_voltspec(); | | |
| 844 | | int pel_replay(); | | |
| 845 | | int pel_loop_play(); | | |
| 846 | | int pel_eval_soft_drivers(); | | |
| 847 | | int pel_eval_dacs_adcs(); | | |
| 848 | | static LM_DIAG_MENU_ITEM menu_list[] = | | |
| 849 | | { | | |
| 850 | | { | | |
| 851 | | { | | |
| 852 | | "1", | | |
| 853 | | "Enter and play patterns", | | |
| 854 | | pel_type_pattern_string, | | |
| 855 | | LM_DIAG_utility, | | |
| 856 | | LM_DIAG_null, | | |
| 857 | | 0 | | |
| 858 | | }, | | |
| 859 | | { | | |
| 860 | | { | | |
| 861 | | "2", | | |
| 862 | | "Set Timing", | | |
| 863 | | pel_set_timing, | | |
| 864 | | LM_DIAG_utility, | | |
| 865 | | LM_DIAG_null, | | |
| 866 | | 0 | | |
| 867 | | }, | | |
| 868 | | { | | |
| 869 | | { | | |
| 870 | | "3", | | |
| 871 | | "Set Voltspec", | | |
| 872 | | pel_set_voltspec, | | |
| 873 | | LM_DIAG_utility, | | |
| 874 | | LM_DIAG_null, | | |
| 875 | | 0 | | |
| 876 | | }, | | |
| 877 | | { | | |
| 878 | | { | | |
| 879 | | "4", | | |
| 880 | | "Play", | | |
| 881 | | pel_replay, | | |
| 882 | | LM_DIAG_utility, | | |
| 883 | | LM_DIAG_null, | | |
| 884 | | 0 | | |
| 885 | | }, | | |
| 886 | | { | | |
| 887 | | { | | |
| 888 | | "5", | | |
| 889 | | "Loop play", | | |
| 890 | | pel_loop_play, | | |
| 891 | | LM_DIAG_utility, | | |
| 892 | | LM_DIAG_null, | | |
| 893 | | 0 | | |
| 894 | | }, | | |
| 895 | | { | | |
| 896 | | { | | |
| 897 | | "6", | | |
| 898 | | "Evaluate Soft-Drivers", | | |
| 899 | | pel_eval_soft_drivers, | | |
| 900 | | LM_DIAG_utility, | | |
| 901 | | LM_DIAG_null, | | |
| 902 | | 0 | | |
| 903 | | }, | | |
| 904 | | { | | |
| 905 | | { | | |
| 906 | | "7", | | |
| 907 | | "Evaluate DACs and ADCs", | | |
| 908 | | pel_eval_dacs_adcs, | | |
| 909 | | LM_DIAG_utility, | | |
| 910 | | LM_DIAG_null, | | |
| 911 | | 0 | | |
| 912 | | }, | | |
| 913 | | }, | | |
| 914 | | static LM_DIAG_MENU menu = | | |
| 915 | | { | | |
| 916 | | "PEL UTILITIES", | | |
| 917 | | sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM), | | |
| 918 | | 0, | | |
| 919 | | menu_list | | |
| 920 | | }; | | |
| 921 | | menu.title = parent_menu->current_selection.menu_text; | | |
| 922 | | menu.items[parent_menu->current_selection].menu_text; | | |
| 923 | | return lm_display_menu(menu); | | |
| 924 | | } | | |
| 925 | | display_pel_errors() | | |
| 926 | | { | | |
| 927 | | if (pel_check_errors(current_lame, current_pel, lm_message) == SUCCESS) { | | |
| 928 | | lm_message("No errors on P1a Electronics Module %ld\n", current_pel); | | |
| 929 | | } | | |
| 930 | | return(SUCCESS); | | |
| 931 | | } | | |
| 932 | | pel_check_errors(lame, slot, prtica) | | |
| 933 | | int lame, slot; | | |
| 934 | | int (*prtica)(); | | |
| 935 | | { | | |
| 936 | | register PEL *pel = (PEL *) (pel_addr(lame, slot)); | | |
| 937 | | int errors = 0; | | |
| 938 | | int i; | | |
| 939 | | { | | |
| 940 | | if (!pel->car.bit.errorL) { | | |
| 941 | | if (pel->car.bit.present && !pel->car.bit.active) { | | |
| 942 | | prtica("PEL error: DAB inserted\n"); | | |
| 943 | | errors++; | | |
| 944 | | } | | |
| 945 | | if (pel->car.bit.present && pel->car.bit.initialize) { | | |
| 946 | | prtica("PEL error: DAB removed\n"); | | |
| 947 | | errors++; | | |
| 948 | | } | | |
| 949 | | if (!pel->car.bit.play_errorL) { | | |
| 950 | | prtica("PEL error: Playing to an uninitialized PEL\n"); | | |
| 951 | | errors++; | | |
| 952 | | } | | |
| 953 | | if (!pel->car.bit.magic_errorL && !pel->car.bit.magic_error_enableL) { | | |
| 954 | | for (i = 0; i < 5; i++) { | | |
| 955 | | if (pel->magic_chip[i].m.parity_out) { | | |
| 956 | | prtica("PEL error: MAGIC(%ld) parity error\n", i); | | |
| 957 | | errors++; | | |
| 958 | | } | | |
| 959 | | } | | |
| 960 | | } | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel.c | DATE 5/23/89 | PAGE # 9/102 |
|--|--|---|--------------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 961 | | for (i = 0; i < 5; i++) { | /* MAGIC shorts */ | |
| 962 | | if (pel->magic_chip[i].m.short_sample) { | | |
| 963 | | printf("PEL error: MAGIC[%ld] short error = %04x\n", | | |
| 964 | | i, pel->magic_chip[i].m.short_sample); | | |
| 965 | | errors++; | | |
| 966 | | } | | |
| 967 | | | | |
| 968 | | if (!errors) { | /* unknown */ | |
| 969 | | printf("PEL error: unknown error\n"); | | |
| 970 | | errors++; | | |
| 971 | | } | | |
| 972 | | } | | |
| 973 | | | | |
| 974 | | return errors ? FAILURE : SUCCESS; | | |
| 975 | | } | | |
| 976 | | | | |
| 977 | | | | |
| 978 | | | | |
| 979 | | pel_count_ddabs() | | |
| 980 | | { | | |
| 981 | | int lane_no, slot_no; | | |
| 982 | | int count = 0; | | |
| 983 | | | | |
| 984 | | if (diag_tme_reset(TRUE) != SUCCESS) { | | |
| 985 | | (void)pel_error("Backplane reset failed during lane probe.\n"); | | |
| 986 | | return 0; | | |
| 987 | | } | | |
| 988 | | for (lane_no = 0; lane_no < NUMBER_OF_LANES; lane_no++) { | | |
| 989 | | if (probe_pac(lane_no) == SUCCESS) { | | |
| 990 | | for (slot_no = 0; slot_no < NUMBER_OF_SLOTS; slot_no++) { | | |
| 991 | | if (probe_pel(lane_no, slot_no) == SUCCESS) { | | |
| 992 | | if (what_dab(lane_no, slot_no) == DIAG_DAB) { | | |
| 993 | | ++count; | | |
| 994 | | } | | |
| 995 | | } | | |
| 996 | | } | | |
| 997 | | | | |
| 998 | | return count; | | |
| 999 | | } | | |
| 1000 | | | | |
| 1001 | | | | |
| 1002 | | pel_error(s) | | |
| 1003 | | char *s; | | |
| 1004 | | { | | |
| 1005 | | (void)ls_error("Lane to Pel %d: %s", | | |
| 1006 | | current_lane + 'A', current_pel, s); | | |
| 1007 | | } | | |
| 1008 | | } | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel_analog.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:25 pm | 1/103 |

```

SOURCE TEXT
LINE #
1  /* SCCS ID: pel_analog.c rev 3.1, 4/24/89 at 07:49:46 */
2
3  .....
4  pel_analog.c
5  .....
6  PEL DAC/ADC tests
7  used in PEL diagnostics
8  .....
9
10 #include <math.h>
11 #include "common.h"
12 #include "lm_diags.h"
13 #include "vrtx.h"
14 #include "msg.h"
15 #include "sccs_def.h"
16 #include "modeler_math.h"
17 #include "magic.h"
18 #include "pel.h"
19
20 int set_vlogl_dac();
21 int set_vlogh_dac();
22 int set_vlth_dac();
23 int set_vah_dac();
24 int set_vlth_dac();
25 int set_val_dac();
26
27 #define DUTVCC_min 3.0
28 #define DUTVCC_max 5.25
29 #define VBI_NO_DAB 0.6
30 #define DUT_ANALOG_ERROR 0.2
31 #define ADC_LSB (5.0 / 256.0)
32 #define DUTVCC_ADC_LSB (6.0 / 256.0)
33 #define DAC_ADC_ERROR 0.140
34
35 /* The following formulas convert floating point voltage setting boundaries */
36 /* based on a floating point DUT Vcc value */
37 /* X = DUT VCC (float); Y = DUT VCC (float) */
38 #define VLOGL_max(X) (X)
39 #define VLOGH_max(X) (X)
40 #define VLTH_max(X) (X)
41 #define VBI_max(X) ((X)/5.0)
42 #define VBI_min(X) ((X)/5.0)
43 #define VBI_max(X) ((X)/5.0)
44 #define VBI_min(X) ((X)/5.0)
45 #define VBI_max(X) ((X)/5.0)
46 #define VBI_min(X) ((X)/5.0)
47 #define VBI_max(X) ((X)/5.0)
48 #define VBI_min(X) ((X)/5.0)
49 #define VBI_max(X) ((X)/5.0)
50
51 /* The following formulas convert a floating point voltage setting into */
52 /* an 8-bit DAC value based on a floating point DUT Vcc value */
53 /* X = input value to set (float); Y = DUT VCC (float) */
54 #define DAC_value_vlth(X,Y) (((X) == 0.0)?0:(u_char)((((X)/(Y)-0.5)*2.0*256.0)))
55 #define DAC_value_vah(X,Y) DAC_value_vlth(X,Y)
56 #define DAC_value_vlth(X,Y) (((X) == 0.0)?0:(u_char)((((X)/(Y)-0.5)*2.0*256.0)))
57 #define DAC_value_val(X,Y) DAC_value_vlth(X,Y)
58 #define DAC_value_vlogl(X,Y) (((X) == 0.0)?0:(u_char)((((X)/(Y)-0.5)*2.0*256.0)))
59 #define DAC_value_vlogh(X,Y) DAC_value_vlogl(X,Y)
60
61 /* The following formulas convert an 8-bit DAC value into a floating point */
62 /* voltage setting based on a floating point DUT Vcc value */
63 /* X = DAC value to set (u_char); Y = DUT VCC (float) */
64 #define float_value_vlth(X,Y) (((float)(X))/(2.0*256.0)+0.5)*(Y)
65 #define float_value_vah(X,Y) float_value_vlth(X,Y)
66 #define float_value_vlth(X,Y) (((float)(X))/(2.0*256.0)+0.5)*(Y)
67 #define float_value_val(X,Y) float_value_vlth(X,Y)
68 #define float_value_vlogl(X,Y) (((float)(X))/(2.0*256.0))*(Y)
69 #define float_value_vlogh(X,Y) float_value_vlogl(X,Y)
70
71 void
72 get_dutvcc(dutvcc)
73 float *dutvcc;
74 {
75     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
76     register u_char raw_dac_value;
77
78     /* Read DAC value twice (hardware requirement) */
79     raw_dac_value = pel->signal_adc;
80     raw_dac_value = pel->signal_adc;
81     *dutvcc = (float)(raw_dac_value) * ADC_LSB;
82 }
83
84 void
85 get_dutvcc(dutvcc)
86 float *dutvcc;
87 {
88     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
89     register u_char raw_dac_value;
90
91     /* Read DAC value twice (hardware requirement) */
92     raw_dac_value = pel->vcc_adc;
93     raw_dac_value = pel->vcc_adc;
94     *dutvcc = (float)(raw_dac_value) * DUTVCC_ADC_LSB;
95 }
96
97 int
98 pel_dac_adc()
99 {
100     int returncode = SUCCESS;
101     float dutvcc, dutanalog;
102
103     get_dutvcc(&dutvcc); /* Read the DUT Vcc line */
104
105     if(dab_type == NO_DAB)
106     {
107         if(dutvcc > DAC_ADC_ERROR)
108         {
109             (void)lm_error("Lane to Pel to DUT Vcc = %.3f exceeds no-DAB maximum.\n",
110                 current_lane + 'A', current_pel, dutvcc);
111             return(FAILURE);
112         }
113     }
114     else
115     {
116         if ((dutvcc > DUTVCC_max) || (dutvcc < DUTVCC_min))
117         {
118             (void)lm_error("DUT Vcc = %.3fV. Legal range is %.3fV to %.3fV.\n",
119                 dutvcc, DUTVCC_min, DUTVCC_max);
120         }
121     }
122 }

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel_analog.c

DATE 5/23/89
TIME 4:41:25 pm

PAGE #
2/104

SOURCE TEXT

```

121     return(FAILURE);
122 }
123
124 if(test_all_dacs(dutvcc, &returacode) != SUCCESS)
125     return(FAILURE);
126
127 if(dab_type == DIAG_DAB)
128 {
129     get_dutanasalog(&dutanasalog);
130     if((dutanasalog > (1.0 + DUT_ANALOG_ERROR)) ||
131        (dutanasalog < (1.0 - DUT_ANALOG_ERROR)))
132     {
133         (void)lm_error("DUTANALOG on Diagnostic DAB = %.3f. Expected \
134 1.000 V +/- %.3f V\n", dutanasalog, DUT_ANALOG_ERROR);
135         return(FAILURE);
136     }
137 }
138
139 return(returacode);
140 }
141
142 int
143 test_a_dac(funcptr, charptr, dutvcc, returacode)
144 int (*funcptr)();
145 char *charptr;
146 float dutvcc;
147 int *returacode;
148 {
149     register int i;
150     register test;
151     static u_char value[9] =
152     {
153         0x00,
154         0x80,
155         0x01,
156         0x40,
157         0x02,
158         0x20,
159         0x04,
160         0x10,
161         0x08
162     };
163
164     if(dab_type == NO_DAB)
165     {
166         if(funcptr(0, 0.0) != SUCCESS)
167         {
168             *returacode = FAILURE;
169             if(lm_error("DAC test for %s (no DAB) fails.\n", charptr) != SUCCESS)
170                 return(FAILURE);
171         }
172     }
173     else
174     {
175         for(test = 0; test < 2; ++test) /* Test 0 => walking 0, else walking 1 */
176         {
177             for(i = 0; i < 9; ++i)
178             {
179                 if(funcptr(test == 0 ? value[i] : -value[i], dutvcc) != SUCCESS)
180                 {
181                     *returacode = FAILURE;
182                     if(lm_error("DAC test for %s fails.\n", charptr) != SUCCESS)
183                         return(FAILURE);
184                 }
185             }
186         }
187     }
188     return(SUCCESS);
189 }
190
191 int
192 test_all_dacs(dutvcc, returacode)
193 float dutvcc;
194 int *returacode;
195 {
196     if(test_a_dac(set_vlogl_dac, "vlogl", dutvcc, returacode) != SUCCESS)
197         return(FAILURE);
198     if(test_a_dac(set_vlogh_dac, "vlogh", dutvcc, returacode) != SUCCESS)
199         return(FAILURE);
200     if(test_a_dac(set_vhth_dac, "vhth", dutvcc, returacode) != SUCCESS)
201         return(FAILURE);
202     if(test_a_dac(set_vsh_dac, "vsh", dutvcc, returacode) != SUCCESS)
203         return(FAILURE);
204     if(test_a_dac(set_vlth_dac, "vlth", dutvcc, returacode) != SUCCESS)
205         return(FAILURE);
206     if(test_a_dac(set_val_dac, "val", dutvcc, returacode) != SUCCESS)
207         return(FAILURE);
208     return(SUCCESS);
209 }
210
211 int
212 check_dac_value(value, min, max)
213 float value;
214 float min;
215 float max;
216 {
217     if ((value > max) || (value < min))
218     {
219         (void)lm_error("Desired DAC setting = %.3f out of range (%.3f - %.3f)\n",
220            value, min, max);
221         return(FAILURE);
222     }
223     return(SUCCESS);
224 }
225
226 int
227 set_vlogl_dac(dac_value, dutvcc)
228 u_char dac_value;
229 float dutvcc;
230 {
231     register PEL *pel = (PEL *) (pel_addr(current_lase, current_pel));
232     float value;
233
234     /* Compute the expected floating point value */
235     value = Float_value_vlogl(dac_value, dutvcc);
236     if(set_dac_and_compare(&pel->vlogl_dac, &pel->vlogl_adc, dac_value, value) !=
237        SUCCESS)
238     {
239         (void)pel_error("Failure setting vlogl DAC.\n");
240         return(FAILURE);

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_analog.c | DATE 5/23/89 TIME 4:41:25 pm | PAGE # 3/105 |
|--|---|--------------------------------------|---------------------------------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 241 | } | | | |
| 242 | return(SUCCESS); | | | |
| 243 | } | | | |
| 244 | } | | | |
| 245 | int | | | |
| 246 | set_vlogl(value) | | | |
| 247 | float value; | | | |
| 248 | { | | | |
| 249 | float dutvcc; | | | |
| 250 | { | | | |
| 251 | get_dutvcc(&dutvcc); /* Read the DUT Vcc line */ | | | |
| 252 | if(check_dac_value(value, VLOGL_min(dutvcc), VLOGL_max(dutvcc)) != SUCCESS) | | | |
| 253 | { | | | |
| 254 | (void)pel_error("Unable to set vlogl to desired value.\n"); | | | |
| 255 | return(FAILURE); | | | |
| 256 | } | | | |
| 257 | return(set_vlogl_dac(DAC_value_vlogl(value, dutvcc), dutvcc)); | | | |
| 258 | } | | | |
| 259 | } | | | |
| 260 | int | | | |
| 261 | set_vlogh_dac(dac_value, dutvcc) | | | |
| 262 | u_char dac_value; | | | |
| 263 | float dutvcc; | | | |
| 264 | { | | | |
| 265 | register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel)); | | | |
| 266 | float value; | | | |
| 267 | { | | | |
| 268 | /* Compute the expected floating point value */ | | | |
| 269 | value = float_value_vlogh(dac_value, dutvcc); | | | |
| 270 | if(set_dac_and_compare(&pel->vlogh_dac, &pel->vlogh_adc, dac_value, value) != | | | |
| 271 | SUCCESS) | | | |
| 272 | { | | | |
| 273 | (void)pel_error("Failure setting vlogh DAC.\n"); | | | |
| 274 | return(FAILURE); | | | |
| 275 | } | | | |
| 276 | return(SUCCESS); | | | |
| 277 | } | | | |
| 278 | } | | | |
| 279 | int | | | |
| 280 | set_vlogh(value) | | | |
| 281 | float value; | | | |
| 282 | { | | | |
| 283 | float dutvcc; | | | |
| 284 | { | | | |
| 285 | get_dutvcc(&dutvcc); /* Read the DUT Vcc line */ | | | |
| 286 | if(check_dac_value(value, VLOGH_min(dutvcc), VLOGH_max(dutvcc)) != SUCCESS) | | | |
| 287 | { | | | |
| 288 | (void)pel_error("Unable to set vlogh to desired value.\n"); | | | |
| 289 | return(FAILURE); | | | |
| 290 | } | | | |
| 291 | return(set_vlogh_dac(DAC_value_vlogh(value, dutvcc), dutvcc)); | | | |
| 292 | } | | | |
| 293 | } | | | |
| 294 | int | | | |
| 295 | set_vlth_dac(dac_value, dutvcc) | | | |
| 296 | u_char dac_value; | | | |
| 297 | float dutvcc; | | | |
| 298 | { | | | |
| 299 | register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel)); | | | |
| 300 | float value; | | | |
| 301 | { | | | |
| 302 | /* Compute the expected floating point value */ | | | |
| 303 | value = float_value_vlth(dac_value, dutvcc); | | | |
| 304 | if(set_dac_and_compare(&pel->vlth_dac, &pel->vlth_adc, dac_value, value) != | | | |
| 305 | SUCCESS) | | | |
| 306 | { | | | |
| 307 | (void)pel_error("Failure setting vlth DAC.\n"); | | | |
| 308 | return(FAILURE); | | | |
| 309 | } | | | |
| 310 | return(SUCCESS); | | | |
| 311 | } | | | |
| 312 | } | | | |
| 313 | int | | | |
| 314 | set_vlth(value) | | | |
| 315 | float value; | | | |
| 316 | { | | | |
| 317 | float dutvcc; | | | |
| 318 | { | | | |
| 319 | get_dutvcc(&dutvcc); /* Read the DUT Vcc line */ | | | |
| 320 | if(check_dac_value(value, VLTH_min(dutvcc), VLTH_max(dutvcc)) != SUCCESS) | | | |
| 321 | { | | | |
| 322 | (void)pel_error("Unable to set vlth to desired value.\n"); | | | |
| 323 | return(FAILURE); | | | |
| 324 | } | | | |
| 325 | return(set_vlth_dac(DAC_value_vlth(value, dutvcc), dutvcc)); | | | |
| 326 | } | | | |
| 327 | } | | | |
| 328 | int | | | |
| 329 | set_val_dac(dac_value, dutvcc) | | | |
| 330 | u_char dac_value; | | | |
| 331 | float dutvcc; | | | |
| 332 | { | | | |
| 333 | register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel)); | | | |
| 334 | float value; | | | |
| 335 | { | | | |
| 336 | /* Compute the expected floating point value */ | | | |
| 337 | value = float_value_val(dac_value, dutvcc); | | | |
| 338 | if(set_dac_and_compare(&pel->val_dac, &pel->val_adc, dac_value, value) != | | | |
| 339 | SUCCESS) | | | |
| 340 | { | | | |
| 341 | (void)pel_error("Failure setting val DAC.\n"); | | | |
| 342 | return(FAILURE); | | | |
| 343 | } | | | |
| 344 | return(SUCCESS); | | | |
| 345 | } | | | |
| 346 | } | | | |
| 347 | int | | | |
| 348 | set_val(value) | | | |
| 349 | float value; | | | |
| 350 | { | | | |
| 351 | float dutvcc; | | | |
| 352 | { | | | |
| 353 | get_dutvcc(&dutvcc); /* Read the DUT Vcc line */ | | | |
| 354 | if(check_dac_value(value, VSL_min(dutvcc), VSL_max(dutvcc)) != SUCCESS) | | | |
| 355 | { | | | |
| 356 | (void)pel_error("Unable to set val to desired value.\n"); | | | |
| 357 | return(FAILURE); | | | |
| 358 | } | | | |
| 359 | return(set_val_dac(DAC_value_val(value, dutvcc), dutvcc)); | | | |
| 360 | } | | | |

[illegible]

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_analog.c | DATE 5/23/89 | PAGE # 4/106 |
|--|--|---|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 361 | | int | | |
| 362 | | set_vhth(dac_value, dutvcc) | | |
| 363 | | u_char dac_value; | | |
| 364 | | float dutvcc; | | |
| 365 | | { | | |
| 366 | | register PEL *pel = (PEL *) (pel_addr(current_lase, current_pel)); | | |
| 367 | | float value; | | |
| 368 | | /* Compute the expected floating point value */ | | |
| 369 | | value = float_value_vhth(dac_value, dutvcc); | | |
| 370 | | if(set_dac_and_compare(spel->vhth_dac, spel->vhth_adc, dac_value, value) != | | |
| 371 | | SUCCESS) | | |
| 372 | | { | | |
| 373 | | (void)pel_error("Failure setting vhth DAC.\n"); | | |
| 374 | | return(FAILURE); | | |
| 375 | | } | | |
| 376 | | return(SUCCESS); | | |
| 377 | | } | | |
| 378 | | int | | |
| 379 | | set_vhth(value) | | |
| 380 | | float value; | | |
| 381 | | { | | |
| 382 | | float dutvcc; | | |
| 383 | | get_dutvcc(&dutvcc); | | |
| 384 | | /* Read the DUT Vcc line */ | | |
| 385 | | if(check_dac_value(value, VTH_min(dutvcc), VTH_max(dutvcc)) != SUCCESS) | | |
| 386 | | { | | |
| 387 | | (void)pel_error("Unable to set vhth to desired value.\n"); | | |
| 388 | | return(FAILURE); | | |
| 389 | | } | | |
| 390 | | return(set_vhth_dac(DAC_value_vhth(value, dutvcc), dutvcc)); | | |
| 391 | | } | | |
| 392 | | int | | |
| 393 | | set_vah(dac_value, dutvcc) | | |
| 394 | | u_char dac_value; | | |
| 395 | | float dutvcc; | | |
| 396 | | { | | |
| 397 | | register PEL *pel = (PEL *) (pel_addr(current_lase, current_pel)); | | |
| 398 | | float value; | | |
| 399 | | u_char adc_value; | | |
| 400 | | /* Compute the expected floating point value */ | | |
| 401 | | if(dac_type == NO_DAB) /* Special case during DAC/ADC test w/o DAB */ | | |
| 402 | | { | | |
| 403 | | /* Read ADC twice (hardware requirement) */ | | |
| 404 | | adc_value = pel->vah_adc; | | |
| 405 | | /* Delay 1s */ | | |
| 406 | | value = ((float)(adc_value - pel->vah_adc)) * ADC_LSB; | | |
| 407 | | /* Compare result against vah without DAB upper bound */ | | |
| 408 | | if(value > (VSH_NO_DAB + DAC_ADC_ERROR)) | | |
| 409 | | { | | |
| 410 | | (void)lm_error("DAC set to 11.3f (%02X). ADC measured 11.3f (%02X).\n", | | |
| 411 | | VSH_NO_DAB, dac_value, value, adc_value); | | |
| 412 | | return(FAILURE); | | |
| 413 | | } | | |
| 414 | | return(SUCCESS); | | |
| 415 | | } | | |
| 416 | | else | | |
| 417 | | value = float_value_vah(dac_value, dutvcc); | | |
| 418 | | if(set_dac_and_compare(spel->vah_dac, spel->vah_adc, dac_value, value) != | | |
| 419 | | SUCCESS) | | |
| 420 | | { | | |
| 421 | | (void)pel_error("Failure setting vah DAC.\n"); | | |
| 422 | | return(FAILURE); | | |
| 423 | | } | | |
| 424 | | return(SUCCESS); | | |
| 425 | | } | | |
| 426 | | int | | |
| 427 | | set_vah(value) | | |
| 428 | | float value; | | |
| 429 | | { | | |
| 430 | | float dutvcc; | | |
| 431 | | get_dutvcc(&dutvcc); | | |
| 432 | | /* Read the DUT Vcc line */ | | |
| 433 | | if(check_dac_value(value, VSH_min(dutvcc), VSH_max(dutvcc)) != SUCCESS) | | |
| 434 | | { | | |
| 435 | | (void)pel_error("Unable to set vah to desired value.\n"); | | |
| 436 | | return(FAILURE); | | |
| 437 | | } | | |
| 438 | | return(set_vah_dac(DAC_value_vah(value, dutvcc), dutvcc)); | | |
| 439 | | } | | |
| 440 | | int | | |
| 441 | | read_adc_and_compare(adcptr, dac_value, expected_value) | | |
| 442 | | u_char *adcptr; | | |
| 443 | | u_char dac_value; | | |
| 444 | | float expected_value; | | |
| 445 | | { | | |
| 446 | | float measured_value, measured_value_II; | | |
| 447 | | u_char adc_value; | | |
| 448 | | /* Read ADC twice (hardware requirement) */ | | |
| 449 | | adc_value = *adcptr; | | |
| 450 | | /* Delay 1s */ | | |
| 451 | | measured_value = ((float)(adc_value - *adcptr)) * ADC_LSB; | | |
| 452 | | /* Delay 1s */ | | |
| 453 | | measured_value_II = ((float)(adc_value - *adcptr)) * ADC_LSB; | | |
| 454 | | /* Compare values */ | | |
| 455 | | if(((measured_value > (expected_value + DAC_ADC_ERROR)) | | |
| 456 | | (measured_value < (expected_value - DAC_ADC_ERROR))) | | |
| 457 | | { | | |
| 458 | | (void)lm_error("DAC set to 11.3f (%02X). ADC first measurement 11.3f (%02X).\n", | | |
| 459 | | expected_value, dac_value, measured_value, adc_value); | | |
| 460 | | (void)lm_error("DAC set to 11.3f (%02X). ADC second measurement 11.3f (%02X).\n", | | |
| 461 | | expected_value, dac_value, measured_value_II, adc_value); | | |
| 462 | | return(FAILURE); | | |
| 463 | | } | | |
| 464 | | return(SUCCESS); | | |
| 465 | | } | | |
| 466 | | int | | |
| 467 | | set_dac_and_compare(dacptr, adcptr, dac_value, expected_value) | | |
| 468 | | u_char *dacptr; | | |
| 469 | | u_char *adcptr; | | |
| 470 | | u_char dac_value; | | |
| 471 | | float expected_value; | | |
| 472 | | { | | |
| 473 | | /* Read ADC twice (hardware requirement) */ | | |
| 474 | | adc_value = *adcptr; | | |
| 475 | | /* Delay 1s */ | | |
| 476 | | measured_value = ((float)(adc_value - *adcptr)) * ADC_LSB; | | |
| 477 | | /* Delay 1s */ | | |
| 478 | | measured_value_II = ((float)(adc_value - *adcptr)) * ADC_LSB; | | |
| 479 | | /* Compare values */ | | |
| 480 | | if(((measured_value > (expected_value + DAC_ADC_ERROR)) | | |

[illegible]

| | | | | |
|--|--|--------------------------------------|-----------------|-----------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_analog.c | DATE 5/23/89 | PAGE # 7/109 |
| TIME 4:41:25 pm | | | | |
| LINE # | SOURCE TEXT | | | |
| 721 | (void)lm_message("%5.3f ", result); | | | |
| 722 | result = pel->val_adc * ADC_LSB; | | | |
| 723 | result = pel->val_adc * ADC_LSB; | | | |
| 724 | (void)lm_message("%5.3f ", result); | | | |
| 725 | (void)lm_message("%5.3f ", Float_value_vth(dac_value,dutvcc)); | | | |
| 726 | | | | |
| 727 | result = pel->vlogl_adc * ADC_LSB; | | | |
| 728 | result = pel->vlogl_adc * ADC_LSB; | | | |
| 729 | (void)lm_message("%5.3f ", result); | | | |
| 730 | result = pel->vlogh_adc * ADC_LSB; | | | |
| 731 | result = pel->vlogh_adc * ADC_LSB; | | | |
| 732 | (void)lm_message("%5.3f ", result); | | | |
| 733 | (void)lm_message("%5.3f ", Float_value_vlogl(dac_value,dutvcc)); | | | |
| 734 | | | | |
| 735 | result = pel->vthh_adc * ADC_LSB; | | | |
| 736 | result = pel->vthh_adc * ADC_LSB; | | | |
| 737 | (void)lm_message("%5.3f ", result); | | | |
| 738 | result = pel->vsh_adc * ADC_LSB; | | | |
| 739 | result = pel->vsh_adc * ADC_LSB; | | | |
| 740 | (void)lm_message("%5.3f ", result); | | | |
| 741 | (void)lm_message("%5.3f ", Float_value_vth(dac_value,dutvcc)); | | | |
| 742 | | | | |
| 743 | (void)lm_message("\n"); | | | |
| 744 | | | | |
| 745 | | | | |

```

Copyright 1989
Logic Modeling Systems
SOURCE PROGRAM
diags/pel_crc.c
DATE 5/23/89 PAGE #
TIME 4:41:26 pm 1/110
SOURCE TEXT
LINE #
1 /* SCDS_ID: pel_crc.c rev 3.1, 4/24/89 at 07:49:50 */
2
3 .....
4 *
5 * CRC routines
6 * used in JCL diagnostics
7 *
8 .....
9
10 #include "common.h"
11 #include "mod_def.h"
12 #include "modeler_extn.h"
13 #include "vrtx.h"
14 #include "ls_diagn.h"
15 #include "Csg.h"
16 #include "Csg_extn.h"
17 #include "csg_def.h"
18 #include "pac.h"
19 #include "pac_def.h"
20 #include "pac_extn.h"
21 #include "magic.h"
22 #include "pel.h"
23
24 #define DDAB_SEL_HIDATA 0x01
25 #define DDAB_NOT_CDE 0x02
26 #define DDAB_NOT_CLR 0x04
27 #define DDAB_CLX 0x08
28
29 #define DDAB_TEST_LODATA_NO_CLK (DDAB_NOT_CLR | DDAB_NOT_CDE)
30 #define DDAB_TEST_LODATA (DDAB_TEST_LODATA_NO_CLK | DDAB_CLX)
31 #define DDAB_TEST_HIDATA (DDAB_TEST_LODATA | DDAB_SEL_HIDATA)
32
33 #define CRC_PREAMBLE_LODATA \
34 "23f5t5SASf5T5T5G5L5Z5S5LU5T5T5T5T5I5I5Ac5Ac5Ac5I6I7z6z7z7"
35 #define CRC_POSTAMBLE_LODATA "U6K7"
36 #define CRC_POSTAMBLE_LO_SIZE (sizeof(CRC_POSTAMBLE_LODATA)/2)
37 #define CRC_PREAMBLE_HIDATA \
38 "23f5t5SASf5T5T5G5L5Z5S5LU5T5T5T5T5I5I5Ac5Ac5Ac5I6I7z6z7z7"
39 #define CRC_POSTAMBLE_HIDATA "O7S7U6K7"
40 #define CRC_POSTAMBLE_HI_SIZE (sizeof(CRC_POSTAMBLE_HIDATA)/2)
41 #define CRC_SHIFT_PATTERN \
42 "23f5t5SASf5T5T5G5L5Z5S5LU5T5T5T5T5I5I5A5A5A5c7U6K7"
43
44 #define FRACK_LOAD "a7c7"
45 #define FRACK_LOAD_SIZE (sizeof(FRACK_LOAD) / 2)
46 #define RISING_0_CRC
47 #define FALLING_0_CRC 0x26
48 #define RISING_512_CRC 0x93
49 #define FALLING_512_CRC 0xd0
50
51 u_long pel_debug = 0;
52
53 generate_preamble(s, chip, byte)
54 register char *s;
55 {
56     register char c;
57
58     c = 'A' + (char)chip;
59     sprintf(s, (byte == 0)? CRC_PREAMBLE_LODATA: CRC_PREAMBLE_HIDATA,
60            c, c, c, c, c);
61     return(strlen(s) >> 1);
62 }
63
64 generate_postamble(s, byte)
65 char *s;
66 {
67     register char *string;
68     register int count;
69
70     if (byte == 0) {
71         string = CRC_POSTAMBLE_LODATA;
72         count = CRC_POSTAMBLE_LO_SIZE;
73     } else {
74         string = CRC_POSTAMBLE_HIDATA;
75         count = CRC_POSTAMBLE_HI_SIZE;
76     }
77     strcpy(s, string);
78     return(count);
79 }
80
81 int
82 set_pel_voltages()
83 {
84     if(set_vlogl(0.8) != SUCCESS) return(FAILURE);
85     if(set_vlogh(2.0) != SUCCESS) return(FAILURE);
86     if(set_vltla(0.3) != SUCCESS) return(FAILURE);
87     if(set_valo(0.4) != SUCCESS) return(FAILURE);
88     if(set_vah(4.0) != SUCCESS) return(FAILURE);
89     if(set_vhth(4.4) != SUCCESS) return(FAILURE);
90     return(SUCCESS);
91 }
92
93
94 /* set up TDC/PAC/PEL/DAB */
95 int
96 pel_diag_dab_test_init(dab_mode)
97 int dab_mode; /* PUBLIC_DAB or PRIVATE_DAB */
98 {
99     if(pel_lane_init() != SUCCESS)
100     {
101         (void)pel_error("Unable to initialize lane for Diagnostic Adapter test.\n");
102         return(FAILURE);
103     }
104     if (pel_dab_init(dab_mode) != SUCCESS)
105     {
106         (void)pel_error("Unable to initialize Diagnostic DAB\n");
107         return(FAILURE);
108     }
109     if(set_pel_voltages() != SUCCESS)
110     {
111         (void)pel_error("Unable to set Pin Electronics DAC voltages.\n");
112         return(FAILURE);
113     }
114     return(SUCCESS);
115 }
116
117 int
118 pel_crc_setup()
119 {
120     if(pel_diag_dab_test_init(PRIVATE_DAB) != SUCCESS)

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel_crc.c

DATE 5/23/89
TIME 4:41:26 pm

PAGE #
2/111

SOURCE TEXT

```

121 {
122     (void)pel_error("Unable to initialize PEL/DAB for CRC test.\n");
123     return(FAILURE);
124 }
125 return(pel_set_diag_dab_period(PAC_MIN_PERIOD * 1000));
126
127
128 int
129 pel_set_diag_dab_period(clock_period)
130 int clock_period; /* in picoseconds */
131 {
132     register int edge;
133     char buffer[80];
134     long edgetime[6]; /* Default edges for Diagnostic Adapter */
135
136     if(clock_period < 200000) /* edge times are close */
137     {
138         edgetime[0] = 10000;
139         edgetime[1] = 10000;
140         edgetime[2] = 10000;
141         edgetime[3] = 10000;
142         edgetime[4] = 10000;
143         edgetime[5] = 0;
144     }
145     else /* edge times are spread out */
146     {
147         edgetime[0] = 500000;
148         edgetime[1] = 500000;
149         edgetime[2] = 500000;
150         edgetime[3] = 1000000;
151         edgetime[4] = 500000;
152         edgetime[5] = 0;
153     }
154
155     if(pel_debug & 2)
156     {
157         diag_get_long((long)clock_period, "Period", 40000L, 8000000000L);
158         for(edge = 0; edge < 6; ++edge)
159         {
160             sprintf(buffer, "Edge %d", edge);
161             diag_get_long((long)edgetime[edge], buffer, 0L, (long)clock_period);
162         }
163     }
164
165     if(tmg_set_timing((long)clock_period, edgetime, 0L, 100000L, 2500000L)
166        != SUCCESS)
167     {
168         (void)lm_error("Unable to set edges (%dps clock period) for CRC test.\n",
169                       clock_period);
170         return(FAILURE);
171     }
172     return(SUCCESS);
173 }
174
175 int
176 pel_crc_test_128K(first_pattern) /* Must run pel_crc_setup before running */
177 int first_pattern;
178 {
179     register int byte;
180     int returncode = SUCCESS;
181
182     for(byte = 0; byte < 2; ++byte)
183     {
184         if(pel_play_crc_sequence(first_pattern, PATTERNS_IN_128K, byte) != SUCCESS)
185         {
186             returncode = FAILURE;
187             if(lm_error("CRC sequence fails while testing %s byte.\n",
188                       byte > "high" : "low") != SUCCESS)
189                 return(FAILURE);
190         }
191     }
192     return(returncode);
193 }
194
195
196 /* PEL CRC Test:
197 */
198 pel_crc()
199 {
200     if(diag_clear_errors() != SUCCESS)
201         return(FAILURE);
202
203     if(pel_crc_setup() != SUCCESS)
204     {
205         (void)pel_error("Unable to perform Pin Electronics CRC test.\n");
206         return(FAILURE);
207     }
208     return(pel_crc_test_128K(0));
209 }
210
211
212 int
213 pel_play_crc_sequence(first_pattern, num_patterns, byte)
214 int first_pattern;
215 int num_patterns;
216 int byte;
217 {
218     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
219     register int pattern_count;
220     register int time_out;
221     register int crc_patterns = num_patterns - BLOCK_SIZE;
222     register int shift;
223     register int chip;
224     register int pass;
225     int i;
226     int magic_0;
227     int crc_result;
228     int returncode = SUCCESS;
229     char amble_buffer[256];
230
231     static u_long right_crc_result[2][10] =
232     {
233         {
234             0x1021D,
235             0x0B46E,
236             0x18A93,
237             0x10AD3,
238             0x04852,
239             0x0D26A,
240
241             /* Results for pass 0 */
242         }
243     }

```


Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pel_crc.c

DATE

5/23/89

PAGE #

TIME

4:41:26 pm

3/112

```

LINE # SOURCE TEXT
241 0x01C0C,
242 0x1948D,
243 0x00977,
244 0x09E4F,
245 },
246 {
247 0x1C106,
248 0x1CBF4,
249 0x1BFD7,
250 0x11323,
251 0x15846,
252 /* Results for pass 1 */
253 0x07924,
254 0x01D45,
255 0x19157,
256 0x03CE9,
257 0x00ACE
258 },
259
260 if((time_out = pac_play()) == 0)
261 {
262 (void)pel_error("Unable to prepare for play.\n");
263 return(FAILURE);
264 }
265
266 for(pass = 0; pass < 2; ++pass)
267 {
268 if(pel_fill_crc(first_pattern + (pass ? BLOCK_SIZE : 0), crc_patterns,
269 current_pel_byte, DOAB_TEST_HIDATA : DOAB_TEST_LODATA, byte,
270 (1st)(SEED << pass)) != SUCCESS)
271 {
272 (void)pel_error("Unable to fill pattern memory with CRC data.\n");
273 return(FAILURE);
274 }
275
276 if(pel_load_pattern_string(CRC_SHIFT_PATTERN, first_pattern +
277 (pass ? 0 : crc_patterns), STOP_MODE) == 0)
278 {
279 (void)pel_error("Unable to load shift pattern string in CRC test.\n");
280 return(FAILURE);
281 }
282
283 (void)lm_message("Start pattern %i, Pass %d, %s Byte: Playing patterns",
284 first_pattern, pass, byte ? "High" : "Low");
285
286 for(chip = 0; chip < 5; ++chip)
287 {
288 lm_message("."); /* Walking dots while messages are on */
289 (void)generate_preamble(amble_buffer, chip, byte);
290 if((build_pattern_data(amble_buffer, first_pattern +
291 (pass ? BLOCK_SIZE : 0)) != 0)
292 {
293 (void)pel_error("Unable to build preamble pattern data.\n");
294 return(FAILURE);
295 }
296
297 pattern_count = generate_postamble(amble_buffer, byte);
298 if((build_pattern_data(amble_buffer, first_pattern +
299 crc_patterns - pattern_count + (pass ? BLOCK_SIZE : 0)) != 0)
300 {
301 (void)pel_error("Unable to build postamble pattern data.\n");
302 return(FAILURE);
303 }
304
305 if((build_pattern_control(first_pattern + (pass ? BLOCK_SIZE : 0),
306 crc_patterns, STOP_MODE) != SUCCESS)
307 {
308 (void)pel_error("Could not build CRC pattern control.\n");
309 return(FAILURE);
310 }
311 if(pac_play(time_out) != SUCCESS)
312 {
313 (void)pel_error("Pattern play failed during CRC tests.\n");
314 return(FAILURE);
315 }
316
317 /* Read the CRC result from magic chip 0 by playing a shift */
318 /* sequence twice and concatenating the results to form a 17 bit */
319 /* checksum */
320 for(crc_result = shift = 0; shift < 3; ++shift)
321 {
322 if(pel_debug & 4) /* Read raw data out of all five magic chips */
323 for(i = 0; i < 5; ++i)
324 {
325 (void)lm_message("Pass %d Byte %d Chip %d Shift %d Magic %d: %04X\n",
326 pass, byte, chip, shift, i, pel->magic_chip[i].reg[11]);
327 }
328
329 /* Make sure control word byte is 0x04 */
330 if(((magic_0 = pel->magic_chip[0].reg[11]) & 0xff00) != 0x0400)
331 {
332 returncode = FAILURE;
333 if(lm_error("Magic chip CRC control byte not equal to 04 \
334 (actual %02X).\n", (magic_0 >> 8) & 0xff)) != SUCCESS)
335 return(FAILURE);
336 }
337
338 crc_result |= ((magic_0 & 0xff) < (8 * shift));
339 if(shift < 2) /* Shift CRC data by playing the shift pattern */
340 {
341 pac_set_first_block(Lane_code(current_lane), (first_pattern +
342 (pass ? 0 : crc_patterns)) / BLOCK_SIZE);
343 if(pac_play(time_out) != SUCCESS)
344 {
345 (void)pel_error("Pattern play failed trying to shift CRC data.\n");
346 return(FAILURE);
347 }
348 }
349 }
350
351 crc_result ^= 0x1ffff;
352
353 /* Compare the actual CRC result with the expected result */
354 if(crc_result != right_crc_result[pass][5 * byte + chip])
355 {
356 returncode = FAILURE;
357 if(lm_error("Pass %d, Byte %d, Chip %d, CRC = %05x. Expected %05x.\n",
358 pass, byte, chip, crc_result, right_crc_result[pass][5 * byte + chip])
359 != SUCCESS)
360 return(FAILURE);

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_crc.c | DATE 5/23/89 | PAGE # 4/113 |
|--|--|--|--------------------|-----------------|
| LINE # | | SOURCE TEXT | TIME 4:41:26 pm | |
| 361 | | if(pel_debug & 1) | | |
| 362 | | msg_play_tty_key(); | | |
| 363 | | } | | |
| 364 | | } | | |
| 365 | | (void)lm_message("done.\n"); | | |
| 366 | | } | | |
| 367 | | return(return_code); | | |
| 368 | | } | | |
| 369 | | } | | |
| 370 | | | | |
| 371 | | int pel_fill_crc(first_patterns, patterns, pel_no, control, dbyte, seed) | | |
| 372 | | /* | | |
| 373 | | INPUT: first_patterns = first patterns number to fill (0's counting) | | |
| 374 | | patterns = number of patterns to fill | | |
| 375 | | pel_no = four bit PEL number code | | |
| 376 | | control = eight bit control word | | |
| 377 | | dbyte : 0 => data in low byte, 1 => data in high byte | | |
| 378 | | seed = random number seed | | |
| 379 | | OUTPUT: return code = SUCCESS or FAILURE | | |
| 380 | | DESCRIPTION: fills pattern memory with pseudorandom | | |
| 381 | | data as well as control bits. Used during PAC/PEL CRC checksum | | |
| 382 | | diagnostics. | | |
| 383 | | */ | | |
| 384 | | | | |
| 385 | | int | | |
| 386 | | pel_fill_crc(first_patterns, patterns, pel_no, control, dbyte, seed) | | |
| 387 | | int first_patterns; | | |
| 388 | | int patterns; | | |
| 389 | | int pel_no; | | |
| 390 | | int control; | | |
| 391 | | int dbyte; | | |
| 392 | | int seed; | | |
| 393 | | { | | |
| 394 | | register int cbyte = (dbyte == 0); /* cbyte is opposite dbyte */ | | |
| 395 | | register u_long *memptr; | | |
| 396 | | register u_long i; | | |
| 397 | | register u_long fill_mask; | | |
| 398 | | register u_long fill_data; | | |
| 399 | | register u_long random_number; | | |
| 400 | | u_long base_address; | | |
| 401 | | u_long bank_offset; | | |
| 402 | | int bank; | | |
| 403 | | { | | |
| 404 | | if(pac_fill_check(first_patterns, patterns) != SUCCESS) | | |
| 405 | | return(FAILURE); | | |
| 406 | | | | |
| 407 | | base_address = pac[current_lase].lase_offset + 4 * first_patterns; | | |
| 408 | | random_number = seed; | | |
| 409 | | | | |
| 410 | | /* Compute fill mask and fill data for banks 0 and 1 (data only) */ | | |
| 411 | | fill_mask = 0x1f << ((8 * cbyte)); | | |
| 412 | | fill_mask = fill_mask << 16; | | |
| 413 | | fill_mask = ~fill_mask; | | |
| 414 | | fill_data = ((u_long)control & 0xff) << ((8 * cbyte)); | | |
| 415 | | fill_data = fill_data << 16; | | |
| 416 | | for(bank = 0, bank_offset = 0, bank < 3; ++bank) | | |
| 417 | | { | | |
| 418 | | if(bank == 2) /* modify fill_mask and fill_data */ | | |
| 419 | | { | | |
| 420 | | fill_mask = (fill_mask 0xffff) & ~0x7fff; | | |
| 421 | | fill_data = (fill_data & 0xffff) ((u_long)pel_no & 0xf) (0x7 << 4); | | |
| 422 | | } | | |
| 423 | | memptr = (u_long *) (base_address + bank_offset); | | |
| 424 | | if(patterns > 1) | | |
| 425 | | { | | |
| 426 | | i = patterns - 1; /* do loop for speed */ | | |
| 427 | | do | | |
| 428 | | { | | |
| 429 | | random_number = Pac_get_random(random_number); | | |
| 430 | | *memptr++ = (random_number & fill_mask) fill_data; | | |
| 431 | | } while (--i); | | |
| 432 | | | | |
| 433 | | *memptr++ = fill_data; | | |
| 434 | | bank_offset += PAT_WORD_BANK; | | |
| 435 | | } | | |
| 436 | | return(SUCCESS); | | |
| 437 | | } | | |
| 438 | | | | |
| 439 | | | | |
| 440 | | /* | | |
| 441 | | int pel_build_crc_check(pel_no, check_edge, seq_length) | | |
| 442 | | /* | | |
| 443 | | INPUT: pel_no = four bit PEL number code | | |
| 444 | | check_edge = branch code (BRANCH_FALLING or BRANCH_RISING) | | |
| 445 | | seq_length = 0, 3, or 512 | | |
| 446 | | OUTPUT: returns number of blocks written (0 => FAILURE) | | |
| 447 | | DESCRIPTION: Builds feedback sequence in first several blocks | | |
| 448 | | of pattern memory. | | |
| 449 | | */ | | |
| 450 | | int | | |
| 451 | | pel_build_crc_check(pel_no, check_edge, seq_length) | | |
| 452 | | int pel_no; | | |
| 453 | | int check_edge; | | |
| 454 | | int seq_length; | | |
| 455 | | { | | |
| 456 | | PAT_WORD pattern_word, *patptr = &pattern_word; | | |
| 457 | | u_long *memptr; | | |
| 458 | | int pattern_no; | | |
| 459 | | int word; | | |
| 460 | | int i; | | |
| 461 | | int blocks_filled; | | |
| 462 | | /* Fill first two blocks of pattern memory with 0's in the lower byte */ | | |
| 463 | | /* of each magic chip pattern word and diag DAB control in the other */ | | |
| 464 | | /* (the control word does not contain a clock bit) */ | | |
| 465 | | if(pel_fill_crc(0, 2 * BLOCK_SIZE, pel_no, DAB_TEST_LOADNO_CLK, 0, 0) | | |
| 466 | | != SUCCESS) | | |
| 467 | | { | | |
| 468 | | (void)pel_error("Unable to fill first two blocks in pattern memory.\n"); | | |
| 469 | | return(FAILURE); | | |
| 470 | | } | | |
| 471 | | memptr = (u_long *) Pattern_to_address(current_lase, 2, 0); | | |
| 472 | | switch(seq_length) | | |
| 473 | | { | | |
| 474 | | case 0: /* Place branch (no clock command) every 5 locations */ | | |
| 475 | | for(i = 0; i < (BLOCK_SIZE / 4); ++i) | | |
| 476 | | { | | |
| 477 | | *memptr++ = 7; | | |
| 478 | | *memptr++ = check_edge; | | |
| 479 | | } | | |
| 480 | | break; | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_crc.c | DATE 5/23/89 TIME 4:41:26 pm | PAGE # 5/114 |
|--|---|-----------------------------------|---------------------------------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 481 | /* Place clock command and branch every 8 locations */ | | | |
| 482 | pattern_no = 0; | | | |
| 483 | while(pattern_no < (2 * BLOCK_SIZE)) | | | |
| 484 | { | | | |
| 485 | (void)pac_read_pattern(pattern_no, patptr); | | | |
| 486 | for(word = 0; word < 5; ++word) | | | |
| 487 | patptr->pattern_data[word] = DDAB_TEST_LODATA << 8; | | | |
| 488 | (void)pac_write_pattern(pattern_no, patptr); | | | |
| 489 | numptr += 7; | | | |
| 490 | numptr += feedback_edge; | | | |
| 491 | pattern_no += 8; | | | |
| 492 | } | | | |
| 493 | break; | | | |
| 494 | /* Place one clock command and branch near end of block */ | | | |
| 495 | pattern_no = (2 * BLOCK_SIZE) - 8; | | | |
| 496 | (void)pac_read_pattern(pattern_no, patptr); | | | |
| 497 | for(word = 0; word < 5; ++word) | | | |
| 498 | patptr->pattern_data[word] = DDAB_TEST_LODATA << 8; | | | |
| 499 | (void)pac_write_pattern(pattern_no, patptr); | | | |
| 500 | numptr += (2 * BLOCK_SIZE) - 1; | | | |
| 501 | numptr += feedback_edge; | | | |
| 502 | break; | | | |
| 503 | default: | | | |
| 504 | (void)pel_error("Software failure - feedback sequence length invalid.\n"); | | | |
| 505 | return(0); | | | |
| 506 | } | | | |
| 507 | /* Find out size of PMM 00 and copy first two blocks appropriate number */ | | | |
| 508 | /* of times */ | | | |
| 509 | switch(pac_get_pmm_size(current_lane, 0)) | | | |
| 510 | { | | | |
| 511 | case PATTERNS_IN_128K: | | | |
| 512 | return(2); | | | |
| 513 | case PATTERNS_IN_512K: | | | |
| 514 | blocks_filled = 4; | | | |
| 515 | break; | | | |
| 516 | case PATTERNS_IN_3M: | | | |
| 517 | blocks_filled = 8; | | | |
| 518 | break; | | | |
| 519 | default: | | | |
| 520 | (void)pel_error("Pattern memory size not valid.\n"); | | | |
| 521 | return(0); | | | |
| 522 | } | | | |
| 523 | /* Now copy the first two blocks (blocks_filled/2); number of times */ | | | |
| 524 | for(i = 0; i <= (blocks_filled >> 3); ++i) | | | |
| 525 | { | | | |
| 526 | if(pel_copy_pat_mem(0, (2 * BLOCK_SIZE) << 1, (2 * BLOCK_SIZE) << 1) | | | |
| 527 | != SUCCESS) | | | |
| 528 | { | | | |
| 529 | (void)pel_error("Unable to copy pattern memory.\n"); | | | |
| 530 | return(0); | | | |
| 531 | } | | | |
| 532 | } | | | |
| 533 | return(blocks_filled); | | | |
| 534 | } | | | |
| 535 | int | | | |
| 536 | pel_copy_pat_mem(from_pattern, to_pattern, num_patterns) | | | |
| 537 | { | | | |
| 538 | register int from_pattern; | | | |
| 539 | register int to_pattern; | | | |
| 540 | register int num_patterns; | | | |
| 541 | { | | | |
| 542 | PAT_WORD pattern_word, *patptr = spattern_word; | | | |
| 543 | register int pattern_no; | | | |
| 544 | { | | | |
| 545 | if((pac_fill_check(from_pattern, num_patterns) != SUCCESS) | | | |
| 546 | (pac_fill_check(to_pattern, num_patterns) != SUCCESS)) | | | |
| 547 | return(FAILURE); | | | |
| 548 | { | | | |
| 549 | for(pattern_no = 0; pattern_no < num_patterns; ++pattern_no) | | | |
| 550 | { | | | |
| 551 | (void)pac_read_pattern(pattern_no + from_pattern, patptr); | | | |
| 552 | (void)pac_write_pattern(pattern_no + to_pattern, patptr); | | | |
| 553 | } | | | |
| 554 | return(SUCCESS); | | | |
| 555 | } | | | |
| 556 | int | | | |
| 557 | pel_feedback_test() | | | |
| 558 | { | | | |
| 559 | PEL *pel = (PEL *) (pel_addr(current_lane, current_pel)); | | | |
| 560 | int returncode = SUCCESS; | | | |
| 561 | int i; | | | |
| 562 | int preamble_length; | | | |
| 563 | int postamble_length; | | | |
| 564 | char amble_buffer[256]; | | | |
| 565 | int num_blocks; | | | |
| 566 | int patterns; | | | |
| 567 | int added_patterns; | | | |
| 568 | int none; | | | |
| 569 | int ptr_offset; | | | |
| 570 | int crc_result; | | | |
| 571 | long *numptr; | | | |
| 572 | int dummy; | | | |
| 573 | { | | | |
| 574 | if(diag_clear_errors() != SUCCESS) | | | |
| 575 | return(FAILURE); | | | |
| 576 | { | | | |
| 577 | /* Initialize the PEL and the Diagnostic DAB */ | | | |
| 578 | if(pel_crc_set() != SUCCESS) | | | |
| 579 | { | | | |
| 580 | (void)pel_error("Unable to perform feedback test.\n"); | | | |
| 581 | return(FAILURE); | | | |
| 582 | } | | | |
| 583 | { | | | |
| 584 | /* First place a feedback sequence starting at block 0 in pattern memory */ | | | |
| 585 | if((num_blocks = pel_build_crc_fbseq(current_pel, BRANCH_FALLING, 0)) == 0) | | | |
| 586 | return(FAILURE); | | | |
| 587 | { | | | |
| 588 | /* Next, place the CRC preamble and postamble in pattern memory */ | | | |
| 589 | /* Set the memory pointer to the beginning of pattern memory, bank 2 */ | | | |
| 590 | numptr = (long *) pattern_to_address(current_lane, 2, 0); | | | |
| 591 | /* First, add the preamble */ | | | |
| 592 | preamble_length = generate_preamble(amble_buffer, 0, 0); | | | |
| 593 | if(build_pattern_data(amble_buffer, num_blocks + BLOCK_SIZE) == 0) | | | |
| 594 | { | | | |
| 595 | (void)pel_error("Unable to build preamble pattern data.\n"); | | | |
| 596 | return(FAILURE); | | | |
| 597 | } | | | |
| 598 | { | | | |
| 599 | /* Now we add patterns to write a one into the CRC circuit */ | | | |
| 600 | if(build_pattern_data(FRAME_LOAD, (num_blocks + BLOCK_SIZE) + | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_crc.c | DATE 5/23/89 | PAGE # 6/115 |
|--|--|--|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 601 | | preamble_length == 0) | | |
| 602 | | { | | |
| 603 | | (void)pel_error("Unable to add feedback load patterns to preamble.\n"); | | |
| 604 | | return(FAILURE); | | |
| 605 | | } | | |
| 606 | | preamble_length += FBANK_LOAD_SIZE; | | |
| 607 | | /* Add at least 8 NOP patterns to end of preamble */ | | |
| 608 | | added_patterns = 8; | | |
| 609 | | /* Add a branch always command to the preamble block */ | | |
| 610 | | ptr_offset = preamble_length + added_patterns - BRANCH_LATENCY; | | |
| 611 | | if((sops = 1 - ptr_offset) > 0) | | |
| 612 | | { | | |
| 613 | | added_patterns += sops; | | |
| 614 | | ptr_offset += sops; | | |
| 615 | | } | | |
| 616 | | if((ptr_offset & 1) == 0) | | |
| 617 | | { | | |
| 618 | | ++added_patterns; | | |
| 619 | | ++ptr_offset; | | |
| 620 | | } | | |
| 621 | | /* Add NOP patterns to end of preamble */ | | |
| 622 | | if(pel_fill_crc((sum_blocks * BLOCK_SIZE) + preamble_length, | | |
| 623 | | added_patterns, current_pel, DDAB_TEST_LODATA_NO_CLK, 0, 0) != SUCCESS) | | |
| 624 | | { | | |
| 625 | | (void)pel_error("Unable to add NOP patterns to preamble.\n"); | | |
| 626 | | return(FAILURE); | | |
| 627 | | } | | |
| 628 | | preamble_length += added_patterns; | | |
| 629 | | *(sumptr + (sum_blocks * BLOCK_SIZE) + ptr_offset) = BRANCH_ALWAYS; | | |
| 630 | | } | | |
| 631 | | /* Now, add the postamble */ | | |
| 632 | | postamble_length = generate_postamble(amble_buffer, 0); | | |
| 633 | | added_patterns = ((ptr_offset + postamble_length - STOP_LATENCY) < 0) ? | | |
| 634 | | - ptr_offset : 0; | | |
| 635 | | if(added_patterns > 0) /* Add NOP patterns to beginning of postamble */ | | |
| 636 | | { | | |
| 637 | | if(pel_fill_crc((sum_blocks + 1) * BLOCK_SIZE, added_patterns, current_pel, | | |
| 638 | | DDAB_TEST_LODATA_NO_CLK, 0, 0) != SUCCESS) | | |
| 639 | | { | | |
| 640 | | (void)pel_error("Unable to add NOP patterns to postamble.\n"); | | |
| 641 | | return(FAILURE); | | |
| 642 | | } | | |
| 643 | | } | | |
| 644 | | if(build_patterns_data(amble_buffer, ((sum_blocks + 1) * BLOCK_SIZE) + | | |
| 645 | | added_patterns) == 0) | | |
| 646 | | { | | |
| 647 | | (void)pel_error("Unable to build postamble pattern data.\n"); | | |
| 648 | | return(FAILURE); | | |
| 649 | | } | | |
| 650 | | postamble_length += added_patterns; | | |
| 651 | | *(sumptr + ((sum_blocks + 1) * BLOCK_SIZE) + ptr_offset + added_patterns) = | | |
| 652 | | STOP; | | |
| 653 | | } | | |
| 654 | | /* Load the link table */ | | |
| 655 | | sumptr = (long *)(&pac[current_lane].lane_offset + LINK_OFFSET); | | |
| 656 | | *sumptr = sum_blocks + 1; | | |
| 657 | | sumptr += sum_blocks; | | |
| 658 | | *sumptr = 0; | | |
| 659 | | } | | |
| 660 | | pac_clock_speed(current_lane, PAC_MIN_PERIOD); | | |
| 661 | | for(i = 0; i < 2; ++i) | | |
| 662 | | { | | |
| 663 | | /* Check for backplane errors */ | | |
| 664 | | if(report_bp_error() != 0) | | |
| 665 | | { | | |
| 666 | | (void)pel_error("Backplane errors - can't initiate play.\n"); | | |
| 667 | | return(FAILURE); | | |
| 668 | | } | | |
| 669 | | pac_set_first_block(Lane_code(current_lane), sum_blocks); | | |
| 670 | | /* Play the pattern (should time out) */ | | |
| 671 | | if(tmg_initiate_play() != SUCCESS) | | |
| 672 | | { | | |
| 673 | | pac_play_cleanup(); | | |
| 674 | | return(FAILURE); | | |
| 675 | | } | | |
| 676 | | lm_delay(1); /* Wait 5ms (plenty of time at 25 MHz) */ | | |
| 677 | | if(bp_mode() != PLAY_MODE) /* Should still be in play mode */ | | |
| 678 | | { | | |
| 679 | | returncode = FAILURE; | | |
| 680 | | if(lm_error("Pattern play completed - expected time out.\n") != SUCCESS) | | |
| 681 | | return(FAILURE); | | |
| 682 | | } | | |
| 683 | | else | | |
| 684 | | { | | |
| 685 | | if(pac_abort_play() != SUCCESS) | | |
| 686 | | { | | |
| 687 | | pac_play_cleanup(); | | |
| 688 | | return(FAILURE); | | |
| 689 | | } | | |
| 690 | | } | | |
| 691 | | if(i == 1) | | |
| 692 | | { | | |
| 693 | | /* Now place a rising feedback sequence in pattern memory */ | | |
| 694 | | if(pel_build_crc_fbck(current_pel, BRANCH_RISING, 0) != sum_blocks) | | |
| 695 | | { | | |
| 696 | | return(FAILURE); | | |
| 697 | | } | | |
| 698 | | } | | |
| 699 | | for(i = 0; i < 2; ++i) | | |
| 700 | | { | | |
| 701 | | /* Place an 8-cycle feedback sequence in pattern memory */ | | |
| 702 | | if(pel_build_crc_fbck(current_pel, 1 ? BRANCH_RISING : BRANCH_FALLING, 8) | | |
| 703 | | != sum_blocks) | | |
| 704 | | { | | |
| 705 | | return(FAILURE); | | |
| 706 | | } | | |
| 707 | | pac_set_first_block(Lane_code(current_lane), sum_blocks); | | |
| 708 | | if(pac_play(TIMEOUT) != SUCCESS) | | |
| 709 | | { | | |
| 710 | | (void)pel_error("Pattern play failed during 8-cycle feedback.\n"); | | |
| 711 | | return(FAILURE); | | |
| 712 | | } | | |
| 713 | | /* Read CRC result and compare with expected */ | | |
| 714 | | crc_result = pel->magic_chip[0].reg[11] & 0xff; | | |
| 715 | | if(crc_result != (1 ? RISING_8_CRC : FALLING_8_CRC)) | | |
| 716 | | { | | |
| 717 | | returncode = FAILURE; | | |
| 718 | | if(lm_error("CRC result does not match. Expected %02X. Actual %02X.\n", | | |
| 719 | | 1 ? RISING_8_CRC : FALLING_8_CRC, crc_result) != SUCCESS) | | |
| 720 | | return(FAILURE); | | |
| 721 | | } | | |
| 722 | | if(pel_set_diag_dab_period(PAC_MAX_PERIOD * 1000) != SUCCESS) | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel_crc.c

DATE 5/23/89
TIME 4:41:26 pm

PAGE #
7/116

```

LINE # SOURCE TEXT
721 {
722 (void)pel_error("Unable to set clock to min frequency in feedback test.\n");
723 return(FAILURE);
724 }
725 Pac_clock_speed(current_lane, PAC_MAX_PERIOD);
726
727 for(i = 0; i < 2; ++i)
728 {
729 /* Place a 512-cycle feedback sequence in pattern memory */
730 if(pel_build_crc_check(current_pel, 1 ? BRANCH_RISING : BRANCH_FALLING, 512)
731 != sum_blocks)
732 return(FAILURE);
733 patterns = preamble_length + postamble_length + BRANCH_LATENCY - 1 +
734 (2 * BLOCK_SIZE) * (1 ? 16 : 17);
735 pac_set_first_block(Lane_code(current_lane), sum_blocks);
736 if(pac_timed_play(PAC_MAX_PERIOD, patterns, &dummy) != SUCCESS)
737 {
738 returncode = FAILURE;
739 if(la_error("Timed pattern play failed during 512-cycle feedback.\n")
740 != SUCCESS)
741 return(FAILURE);
742 }
743 /* Read CRC result and compare with expected */
744 crc_result = pel_magic_chip(0).reg[11] & 0xFF;
745 if(crc_result != (1 ? RISING_512_CRC : FALLING_512_CRC))
746 {
747 returncode = FAILURE;
748 if(la_error("CRC result does not match. Expected %02X. Actual %02X.\n",
749 1 ? RISING_512_CRC : FALLING_512_CRC, crc_result) != SUCCESS)
750 return(FAILURE);
751 }
752 }
753 return(returncode);
754 }
755
756 set_user_bit(patterns_no, value)
757 {
758 PAT_WORD patterns_word;
759
760 (void)pac_read_patterns(patterns_no, &patterns_word);
761 patterns_word.pattern_mask = value;
762 (void)pac_write_patterns(patterns_no, &patterns_word);
763 }
764
765 #define INUSELED_MASK 0x8000
766 #define KEEPALIVE_MASK 0x4000
767 #define CLK10MHz2_MASK 0x2000
768 #define CLK10MHz2_MASK 0x1000
769 #define USER_MASK 0x0000
770 #define KEEPALIVE_PATTERN "2222227"
771 #define MAX_CLK10MHz2_ATTEMPTS 100
772 #define MAX_CLK10MHz2_ATTEMPTS 50
773
774 pel_keepalive_test()
775 {
776 static long edgetime[5] = {
777 100001, 100001, 100001, 100001, 100001
778 };
779 static long period = 400001;
780 int pac_replay();
781 register PEL *pel;
782 u_short patterns[5];
783 int attempts, differences, found_a_0, found_a_1;
784 int errors = 0;
785
786 if(diag_clear_errors() != SUCCESS)
787 return(FAILURE);
788
789 if (set_vlog1(0.8) == FAILURE) return(FAILURE);
790 if (set_vlog2(2.0) == FAILURE) return(FAILURE);
791 if (set_vhth(0.3) == FAILURE) return(FAILURE);
792 if (set_val(0.4) == FAILURE) return(FAILURE);
793 if (set_vah(3.0) == FAILURE) return(FAILURE);
794 if (set_vhth(4.4) == FAILURE) return(FAILURE);
795
796 if (pel_lane_init() != SUCCESS) { /* set up pac/tmg/pel/dab */
797 (void)la_error("Unable to initialize lane to\n", current_lane + "A");
798 return(FAILURE);
799 }
800
801 if(pel_diag_dab_test_init(PUBLIC_DAB) != SUCCESS) {
802 (void)pel_error("Unable to initialize PEL/DAB for keepalive/trigger bit test.\n");
803 return(FAILURE);
804 }
805
806 if (tmg_set_timing(period, edgetime, 01, 1000001, 25000001) != SUCCESS) {
807 return FAILURE;
808 }
809
810 /*
811 1. Verify that nothing is driving bus(16..1)
812 */
813 if (pel_play_pattern_string("2222227", 0) != SUCCESS) {
814 (void)la_error("Unable to play keepalive/trigger bit test patterns.\n");
815 return(FAILURE);
816 }
817
818 patterns[4] = patterns[3] = patterns[2] = patterns[1] = patterns[0] = 0x0000;
819 if (compare_magic_chips(11, patterns) != SUCCESS) { /* VALUE SAMPLE */
820 (void)pel_error("Failed buffer 512 test.\n");
821 errors++;
822 }
823
824 /*
825 2. Verify USER=1, INUSE LED=0, KEEPALIVE=1, CLK 10MHz2=1 and CLK 1MHz2=1
826 */
827 pel->csr.bit.eeprom_sel = 1;
828 pel->csr.bit.in_use_led = 1;
829 if (pel_load_pattern_string(KEEPALIVE_PATTERN, 0, STOP_MODE) == FAILURE) {
830 (void)pel_error("Unable to load keepalive/trigger bit test patterns.\n");
831 return(FAILURE);
832 }
833
834 (void)set_user_bit(strlen(KEEPALIVE_PATTERN)/2 - 1, 1);
835 if(diag_play() != SUCCESS) {

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_crc.c | DATE 5/23/89 | PAGE # 8/117 |
|--|---|-----------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 841 | {void}pel_error("Unable to play keepalive/trigger bit test pattern.\n"); | | | |
| 842 | return(FAILURE); | | | |
| 843 | { | | | |
| 844 | if ((pel->magic_chip[0].m.logic_sample & INUSELED_MASK) != 0) { | | | |
| 845 | (void)pel_error("IN USE LED not functional.\n"); | | | |
| 846 | errors++; | | | |
| 847 | } | | | |
| 848 | if ((pel->magic_chip[0].m.logic_sample & USER_MASK) != USER_MASK) { | | | |
| 849 | (void)pel_error("USER TRIGGER bit not functional.\n"); | | | |
| 850 | errors++; | | | |
| 851 | } | | | |
| 852 | if ((pel->magic_chip[0].m.logic_sample & KEEPALIVE_MASK) != KEEPALIVE_MASK) { | | | |
| 853 | (void)pel_error("KEEPALIVE bit not functional.\n"); | | | |
| 854 | errors++; | | | |
| 855 | } | | | |
| 856 | if ((pel->magic_chip[0].m.logic_sample & CLK1MHZ_MASK) != CLK1MHZ_MASK) { | | | |
| 857 | (void)pel_error("CLK 1MHZ bit not functional.\n"); | | | |
| 858 | errors++; | | | |
| 859 | } | | | |
| 860 | if ((pel->magic_chip[0].m.logic_sample & CLK10MHZ_MASK) != CLK10MHZ_MASK) { | | | |
| 861 | (void)pel_error("CLK 10MHZ bit not functional.\n"); | | | |
| 862 | errors++; | | | |
| 863 | } | | | |
| 864 | } | | | |
| 865 | /* | | | |
| 866 | * 3. Verify USER=0, INUSE LED=1 and KEEPALIVE=0 | | | |
| 867 | */ | | | |
| 868 | pel->car.bit.in_use_led = 0; | | | |
| 869 | if (pel_play_pattern_string("z2z2z2z7", 0) != SUCCESS) { | | | |
| 870 | (void)pel_error("Unable to play keepalive/trigger bit test pattern.\n"); | | | |
| 871 | return(FAILURE); | | | |
| 872 | } | | | |
| 873 | if ((pel->magic_chip[0].m.logic_sample & INUSELED_MASK) != INUSELED_MASK) { | | | |
| 874 | (void)pel_error("IN USE LED not functional.\n"); | | | |
| 875 | errors++; | | | |
| 876 | } | | | |
| 877 | if ((pel->magic_chip[0].m.logic_sample & USER_MASK) != 0) { | | | |
| 878 | (void)pel_error("USER TRIGGER bit not functional.\n"); | | | |
| 879 | errors++; | | | |
| 880 | } | | | |
| 881 | if ((pel->magic_chip[0].m.logic_sample & KEEPALIVE_MASK) != 0) { | | | |
| 882 | (void)pel_error("KEEPALIVE bit not functional.\n"); | | | |
| 883 | errors++; | | | |
| 884 | } | | | |
| 885 | } | | | |
| 886 | /* | | | |
| 887 | * 4. Verify that CLK10 MHz toggles | | | |
| 888 | */ | | | |
| 889 | attempts = 0; | | | |
| 890 | found_a_0 = FALSE; | | | |
| 891 | found_a_1 = FALSE; | | | |
| 892 | while (++attempts <= MAX_CLK10MHZ_ATTEMPTS && (!found_a_0 !found_a_1)) { | | | |
| 893 | if (diag_play()) != SUCCESS) { | | | |
| 894 | (void)pel_error("Unable to play keepalive/trigger bit test pattern.\n"); | | | |
| 895 | return(FAILURE); | | | |
| 896 | } | | | |
| 897 | if ((pel->magic_chip[0].m.ttl_sample & CLK10MHZ_MASK) == 0) { | | | |
| 898 | found_a_0 = TRUE; | | | |
| 899 | } else { | | | |
| 900 | found_a_1 = TRUE; | | | |
| 901 | } | | | |
| 902 | } | | | |
| 903 | if (--attempts == MAX_CLK10MHZ_ATTEMPTS) { | | | |
| 904 | (void)lm_error("CLK 10MHZ bit not functional in %d attempts.\n", attempts); | | | |
| 905 | errors++; | | | |
| 906 | } else { | | | |
| 907 | (void)lm_message("It took %d attempts to verify that CLK 10MHZ toggles.\n", attempts); | | | |
| 908 | } | | | |
| 909 | /* | | | |
| 910 | * 5. Verify that CLK1 MHz is approximately the right frequency | | | |
| 911 | */ | | | |
| 912 | /* first see if CLK 1MHZ toggles */ | | | |
| 913 | attempts = 0; | | | |
| 914 | found_a_0 = FALSE; | | | |
| 915 | found_a_1 = FALSE; | | | |
| 916 | while (++attempts <= MAX_CLK1MHZ_ATTEMPTS && (!found_a_0 !found_a_1)) { | | | |
| 917 | if (diag_play()) != SUCCESS) { | | | |
| 918 | (void)pel_error("Unable to play keepalive/trigger bit test pattern.\n"); | | | |
| 919 | return(FAILURE); | | | |
| 920 | } | | | |
| 921 | if ((pel->magic_chip[0].m.ttl_sample & CLK1MHZ_MASK) == 0) { | | | |
| 922 | found_a_0 = TRUE; | | | |
| 923 | } else { | | | |
| 924 | found_a_1 = TRUE; | | | |
| 925 | } | | | |
| 926 | } | | | |
| 927 | if (--attempts == MAX_CLK1MHZ_ATTEMPTS) { | | | |
| 928 | (void)lm_error("CLK 1MHZ bit not functional in %d attempts.\n", attempts); | | | |
| 929 | errors++; | | | |
| 930 | } else { | | | |
| 931 | (void)lm_message("It took %d attempts to verify that CLK 1MHZ toggles.\n", attempts); | | | |
| 932 | } | | | |
| 933 | /* | | | |
| 934 | * Measure CLK 1MHZ frequency crudely | | | |
| 935 | */ | | | |
| 936 | /* note: It is important that the keepalive clocks get turned off before they get turned on | | | |
| 937 | * to verify that they start up both quickly and correctly. | | | |
| 938 | */ | | | |
| 939 | attempts = 0; | | | |
| 940 | differences = 0; | | | |
| 941 | for (attempts = 0; attempts < MAX_CLK1MHZ_ATTEMPTS; attempts++) { | | | |
| 942 | pel->car.bit.private = 1; | | | |
| 943 | if (pel_play_pattern_string("z2z2z2z7", 0) != SUCCESS) { /* turn off clocks */ | | | |
| 944 | (void)pel_error("Unable to play keepalive/trigger bit test pattern.\n"); | | | |
| 945 | return(FAILURE); | | | |
| 946 | } | | | |
| 947 | lm_delay(1); /* make keepalive clocks stay off for a while */ | | | |
| 948 | pel->car.bit.private = 0; | | | |
| 949 | if (pel_play_pattern_string("z2z2z2z7", 0) != SUCCESS) { /* turn on clocks */ | | | |
| 950 | (void)pel_error("Unable to play keepalive/trigger bit test pattern.\n"); | | | |
| 951 | return(FAILURE); | | | |
| 952 | } | | | |
| 953 | if ((pel->magic_chip[0].m.ttl_sample & CLK1MHZ_MASK) != (pel->magic_chip[0].m.logic_sample & CLK1MHZ_MASK)) { | | | |
| 954 | differences++; | | | |
| 955 | } | | | |
| 956 | } | | | |

| | | | | |
|--|--|-----------------------------------|-----------------|-----------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_crc.c | DATE 5/23/89 | PAGE # 9/118 |
| TIME 4:41:26 pm | | | | |
| LINE # | SOURCE TEXT | | | |
| 961 | } | | | |
| 962 | } | | | |
| 963 | if ((differences = 100) / attempts > 50) { | | | |
| 964 | pel_error("CLK LHM2 is off frequency\n"); | | | |
| 965 | error(" "); | | | |
| 966 | } | | | |
| 967 | lm_message("For CLK LHM2 found %d differences in %d attempts\n", differences, attempts); | | | |
| 968 | | | | |
| 969 | return errors > FAILURE : SUCCESS; | | | |
| 970 | } | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_edge.c | DATE 5/23/89 | PAGE # 1/119 |
|--|--|------------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SOC5 ID: pel_edge.c rev 3.1, 4/24/89 at 07:49:55 */ | | | |
| 2 | /* | | | |
| 3 | | | | |
| 4 | * EDGE routines | | | |
| 5 | * used in PEL diagnostics | | | |
| 6 | * | | | |
| 7 | | | | |
| 8 | */ | | | |
| 9 | | | | |
| 10 | #include "common.h" | | | |
| 11 | #include "mod_def.h" | | | |
| 12 | #include "moduler_extn.h" | | | |
| 13 | #include "vtx.h" | | | |
| 14 | #include "in_diags.h" | | | |
| 15 | #include "tmg.h" | | | |
| 16 | #include "tmg_extn.h" | | | |
| 17 | #include "pac.h" | | | |
| 18 | #include "pac_def.h" | | | |
| 19 | #include "pac_extn.h" | | | |
| 20 | #include "magic.h" | | | |
| 21 | #include "pel.h" | | | |
| 22 | | | | |
| 23 | | | | |
| 24 | #define EDGE_TEST_PREAMBLE "z2f5z5z5A5f5f5f505z5z5z5lu5a5h5a5h5z5f5f5f5f5f5" | | | |
| 25 | #define EDGE_TEST_PATTERN "z2z2z2f7" | | | |
| 26 | #define EDGE_TEST_PERIOD 100000 | | | |
| 27 | | | | |
| 28 | | | | |
| 29 | /* | | | |
| 30 | * EDGE Tests: | | | |
| 31 | */ | | | |
| 32 | pel_edge() | | | |
| 33 | { | | | |
| 34 | register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel)); | | | |
| 35 | static long edgetime[5] = { | | | |
| 36 | 10000, 20000, 10000, 40000, 50000, 60000 | | | |
| 37 | }; | | | |
| 38 | long tmp; | | | |
| 39 | static long period = EDGE_TEST_PERIOD; | | | |
| 40 | u_short pattern[5]; | | | |
| 41 | short test, edge; | | | |
| 42 | int errors; | | | |
| 43 | | | | |
| 44 | static short result[5] = { | | | |
| 45 | 0xffff, 0xffff, 0x9999, 0x3333, 0x7777, 0xffff}; | | | |
| 46 | | | | |
| 47 | if (diag_clear_errors() != SUCCESS) | | | |
| 48 | return (FAILURE); | | | |
| 49 | | | | |
| 50 | errors = 0; | | | |
| 51 | | | | |
| 52 | if (pel_lane_init() != SUCCESS) { /* set up pec/tmg/pel/dab */ | | | |
| 53 | (void) pel_error("Unable to initialize lane to", current_lane + 'A'); | | | |
| 54 | return (FAILURE); | | | |
| 55 | } | | | |
| 56 | pel->bar.bit.resetl = 0; /* disable backplane errors */ | | | |
| 57 | | | | |
| 58 | if (tmg_set_timing(period, edgetime, 01, 1000001, 25000001) != SUCCESS) { | | | |
| 59 | return FAILURE; | | | |
| 60 | } | | | |
| 61 | | | | |
| 62 | /* | | | |
| 63 | * Set up MAGIC Control register and turn off MAGIC output enables | | | |
| 64 | * | | | |
| 65 | * Repeat for all nibbles: | | | |
| 66 | * DUT(0): R1 SET(2) RESET(0) | | | |
| 67 | * DUT(1): R1 SET(3) RESET(1) | | | |
| 68 | * DUT(2): R1 SET(4) RESET(2) | | | |
| 69 | * DUT(3): R1 SET(5) RESET(3) | | | |
| 70 | * | | | |
| 71 | if (pel_play_pattern_string(EDGE_TEST_PREAMBLE, 0) != SUCCESS) { | | | |
| 72 | (void) pel_error("Failure to play EDGE test preamble"); | | | |
| 73 | return (FAILURE); | | | |
| 74 | } | | | |
| 75 | | | | |
| 76 | for (test = 0; test < 6; test++) { | | | |
| 77 | tmp = edgetime[0]; | | | |
| 78 | edgetime[0] = edgetime[5]; | | | |
| 79 | edgetime[5] = edgetime[4]; | | | |
| 80 | edgetime[4] = edgetime[3]; | | | |
| 81 | edgetime[3] = edgetime[2]; | | | |
| 82 | edgetime[2] = edgetime[1]; | | | |
| 83 | edgetime[1] = tmp; | | | |
| 84 | | | | |
| 85 | if (tmg_set_timing(period, edgetime, 01, 1000001, 25000001) != SUCCESS) { | | | |
| 86 | return (FAILURE); | | | |
| 87 | } | | | |
| 88 | | | | |
| 89 | if (pel_play_pattern_string(EDGE_TEST_PATTERN, 0) != SUCCESS) { | | | |
| 90 | (void) pel_error("Failure to play EDGE test pattern"); | | | |
| 91 | return (FAILURE); | | | |
| 92 | } | | | |
| 93 | | | | |
| 94 | pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] | | | |
| 95 | = result[test]; | | | |
| 96 | if (compare_magic_chips(0, pattern) != SUCCESS) { | | | |
| 97 | errors++; | | | |
| 98 | } | | | |
| 99 | } | | | |
| 100 | | | | |
| 101 | return errors ? FAILURE : SUCCESS; | | | |
| 102 | } | | | |
| 103 | | | | |
| 104 | | | | |
| 105 | | | | |
| 106 | | | | |
| 107 | pattern_word_rotate(pattern_no) | | | |
| 108 | int pattern_no; | | | |
| 109 | { | | | |
| 110 | PAT_WORD pattern_word, *patptr = &pattern_word; | | | |
| 111 | int tmp; | | | |
| 112 | | | | |
| 113 | (void) pac_read_pattern(pattern_no, patptr); | | | |
| 114 | tmp = patptr->pattern.data[0]; | | | |
| 115 | patptr->pattern.data[0] = patptr->pattern.data[1]; | | | |
| 116 | patptr->pattern.data[1] = patptr->pattern.data[2]; | | | |
| 117 | patptr->pattern.data[2] = patptr->pattern.data[3]; | | | |
| 118 | patptr->pattern.data[3] = patptr->pattern.data[4]; | | | |
| 119 | patptr->pattern.data[4] = tmp; | | | |
| 120 | (void) pac_write_pattern(pattern_no, patptr); | | | |

[illegible]

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_edge.c | DATE 5/23/89 | PAGE # 3/121 |
|--|--|--|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 241 | | return_status = FAILURE; | | |
| 242 | | if (lm_error(format, | | |
| 243 | | current_lane + 'A', current_pel, | | |
| 244 | | 0, 1, soft_driver_test->driver) != SUCCESS) { | | |
| 245 | | return(FAILURE); | | |
| 246 | | } | | |
| 247 | | | | |
| 248 | | for (magic = 1; magic < 5; magic++) { | | |
| 249 | | pattern_word_rotate(4); | | |
| 250 | | pattern_word_rotate(5); | | |
| 251 | | pattern_word_rotate(6); | | |
| 252 | | pattern_word_rotate(7); | | |
| 253 | | pattern_word_rotate(8); | | |
| 254 | | pattern_word_rotate(9); | | |
| 255 | | pattern_word_rotate(10); | | |
| 256 | | pattern_word_rotate(11); | | |
| 257 | | pattern_word_rotate(12); | | |
| 258 | | if (diag_play() == FAILURE) { | | |
| 259 | | (void)pel_error("Unable to play driver test pattern.\n"); | | |
| 260 | | return(FAILURE); | | |
| 261 | | } | | |
| 262 | | if (compare_magic_chips(11, pattern) != SUCCESS) { /* value */ | | |
| 263 | | return_status = FAILURE; | | |
| 264 | | if (lm_error(format, | | |
| 265 | | current_lane + 'A', current_pel, | | |
| 266 | | magic, (magic + 1) % 5, | | |
| 267 | | soft_driver_test->driver) != SUCCESS) { | | |
| 268 | | return(FAILURE); | | |
| 269 | | } | | |
| 270 | | } | | |
| 271 | | | | |
| 272 | | lm_message("done.\n"); | | |
| 273 | | | | |
| 274 | | | | |
| 275 | | | | |
| 276 | | /* | | |
| 277 | | * Test all medium drivers | | |
| 278 | | */ | | |
| 279 | | | | |
| 280 | | if (tmg_set_timing(period, edgetime, 01, 1000001, 100000001) != SUCCESS) { | | |
| 281 | | return FAILURE; | | |
| 282 | | } | | |
| 283 | | | | |
| 284 | | lm_message("Testing Medium Drivers"); | | |
| 285 | | for (medium_driver_test = Medium_Driver_Tests, medium_driver_test->driver, ++medium_driver_test) { | | |
| 286 | | lm_message("-"); | | |
| 287 | | pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = medium_driver_test->result; | | |
| 288 | | if (pel_play_pattern_string(medium_driver_test->pattern, 0) != SUCCESS) { | | |
| 289 | | (void)pel_error("Unable to play driver test pattern.\n"); | | |
| 290 | | return (FAILURE); | | |
| 291 | | } | | |
| 292 | | if (compare_magic_chips(11, pattern) != SUCCESS) { /* value */ | | |
| 293 | | return_status = FAILURE; | | |
| 294 | | if (lm_error("Lane 0c pel td: Magic chip 0 failed ts test.\n", | | |
| 295 | | current_lane + 'A', current_pel, | | |
| 296 | | medium_driver_test->driver) != SUCCESS) { | | |
| 297 | | return(FAILURE); | | |
| 298 | | } | | |
| 299 | | } | | |
| 300 | | for (magic = 1; magic < 5; magic++) { | | |
| 301 | | pattern_word_rotate(4); | | |
| 302 | | pattern_word_rotate(5); | | |
| 303 | | pattern_word_rotate(6); | | |
| 304 | | pattern_word_rotate(7); | | |
| 305 | | pattern_word_rotate(8); | | |
| 306 | | pattern_word_rotate(9); | | |
| 307 | | pattern_word_rotate(10); | | |
| 308 | | pattern_word_rotate(11); | | |
| 309 | | pattern_word_rotate(12); | | |
| 310 | | pattern_word_rotate(13); | | |
| 311 | | pattern_word_rotate(14); | | |
| 312 | | pattern_word_rotate(15); | | |
| 313 | | if (diag_play() == FAILURE) { | | |
| 314 | | (void)pel_error("Unable to play driver test pattern.\n"); | | |
| 315 | | return(FAILURE); | | |
| 316 | | } | | |
| 317 | | if (compare_magic_chips(11, pattern) != SUCCESS) { /* value */ | | |
| 318 | | return_status = FAILURE; | | |
| 319 | | if (lm_error("Lane 0c pel td: Magic chip 0 failed ts test.\n", | | |
| 320 | | current_lane + 'A', current_pel, | | |
| 321 | | magic, | | |
| 322 | | medium_driver_test->driver) != SUCCESS) { | | |
| 323 | | return(FAILURE); | | |
| 324 | | } | | |
| 325 | | } | | |
| 326 | | | | |
| 327 | | lm_message("done.\n"); | | |
| 328 | | | | |
| 329 | | | | |
| 330 | | return(return_status); | | |
| 331 | | } | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pel_err.c

DATE

5/23/89

PAGE #

1/122

TIME 4:41:27 pm

```

1  /* SOCS_ID: pel_err.c rev 3.1, 4/24/89 at 07:49:38 */
2
3  #include "common.h"
4  #include "mod_def.h"
5  #include "modeler_extn.h"
6  #include "magic.h"
7  #include "pel.h"
8
9  pel_error_test()
10 {
11     /* This test simulates the insertion and removal of an adapter. */
12     /* On the diagnostic dab, KEIN is connected to PRESENT. */
13     /* If KEIN is high, adapter is removed. */
14     /* If KEIN is low, adapter is inserted. */
15     /* Check 1 (reset):      car = f522 */
16     /* Check 2 (insertion):  car = f502 */
17     /* Check 3 (initialization): car = f512 */
18     /* Check 4 (removal):    car = f132 */
19     /* Check 5 (play error): car = e122 */
20
21     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
22     register long retval = SUCCESS;
23     char *error_format =
24         "%s:playerror=%td error=%td(%td) present=%td(%td) active=%td(%td)\n";
25
26     if (diag_clear_errors() != SUCCESS)
27         return(FAILURE);
28
29     if (pel_lane_init() != SUCCESS) { /* set up pac/tmg/pel/dab */
30         (void) lm_error("Unable to initialize lane %c\n",
31             current_lane + 'A');
32         return(FAILURE);
33     }
34
35     pel->car.reg = 0; /* start in a known (default) state */
36     pel->car.bit.esprun_in = 1; /* diag dab not present */
37     pel_clear_pel_errors();
38     lm_message("Check 1 (reset):      car = %04x\n", pel->car.reg);
39     if ((pel->car.reg) != 0xf522) {
40         (void) lm_error(error_format, "Reset check",
41             pel->car.bit.play_errorL, 1,
42             pel->car.bit.errorL, 1,
43             pel->car.bit.present, 0,
44             pel->car.bit.active, 0);
45         retval = FAILURE;
46     }
47     pel->car.bit.esprun_in = 0; /* diag dab present */
48     lm_message("Check 2 (insertion):  car = %04x\n", pel->car.reg);
49     if ((pel->car.reg) != 0xf502) {
50         (void) lm_error(error_format, "Insertion check",
51             pel->car.bit.play_errorL, 1,
52             pel->car.bit.errorL, 0,
53             pel->car.bit.present, 1,
54             pel->car.bit.active, 0);
55         retval = FAILURE;
56     }
57     pel->car.bit.resetL = 0; /* reset */
58     pel->car.bit.initialize = 0;
59     pel->car.bit.initialize = 1;
60     pel->car.bit.resetL = 1; /* lift reset */
61     lm_message("Check 3 (initialization): car = %04x\n", pel->car.reg);
62     if ((pel->car.reg) != 0xf512) {
63         (void) lm_error(error_format, "Initialization check",
64             pel->car.bit.play_errorL, 1,
65             pel->car.bit.errorL, 1,
66             pel->car.bit.present, 1,
67             pel->car.bit.active, 1);
68         retval = FAILURE;
69     }
70     pel->car.bit.esprun_in = 1; /* diag dab not present */
71     lm_message("Check 4 (removal):    car = %04x\n", pel->car.reg);
72     if ((pel->car.reg) != 0xf132) {
73         (void) lm_error(error_format, "Removal check",
74             pel->car.bit.play_errorL, 1,
75             pel->car.bit.errorL, 0,
76             pel->car.bit.present, 0,
77             pel->car.bit.active, 0);
78         retval = FAILURE;
79     }
80
81     pel->car.bit.initialize = 0;
82     pel->car.bit.resetL = 0; /* reset */
83     pel->car.bit.resetL = 1; /* lift reset */
84     if (pel_play_pattern_string("11111111", 0) != SUCCESS) {
85         lm_message("ERROR playing pattern string\n");
86         return FAILURE;
87     }
88     lm_message("Check 5 (play error):  car = %04x\n", pel->car.reg);
89     if ((pel->car.reg) != 0xe122) {
90         (void) lm_error(error_format, "Play error check",
91             pel->car.bit.play_errorL, 0,
92             pel->car.bit.errorL, 0,
93             pel->car.bit.present, 0,
94             pel->car.bit.active, 0);
95         retval = FAILURE;
96     }
97
98     pel_clear_pel_errors();
99     return(retval);
100 }

```

1099

5,353,243

1100

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_id.c | DATE 5/23/89 TIME 4:41:27 pm | PAGE # 1/123 |
|--|---|----------------------------------|---------------------------------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCDS_ID: pel_id.c rev 3.1, 4/24/89 at 07:50:08 */ | | | |
| 2 | #include "common.h" | | | |
| 3 | #include "lm_diags.h" | | | |
| 4 | #include "id.h" | | | |
| 5 | #include "magic.h" | | | |
| 6 | #include "pel.h" | | | |
| 7 | | | | |
| 8 | /* | | | |
| 9 | * Compute and verify checksum on ID PROM | | | |
| 10 | */ | | | |
| 11 | * Pass pointer to first byte in ID PROM. | | | |
| 12 | * Subsequent ID_NUM_BYTES-1 bytes are 4 bytes apart. | | | |
| 13 | * Algorithm: | | | |
| 14 | * Initialize checksum to an arbitrary (but well-known) value | | | |
| 15 | * for each byte (including checksum) | | | |
| 16 | * circular left shift left checksum | | | |
| 17 | * add data byte | | | |
| 18 | * mask checksum to 8 bits | | | |
| 19 | * Returns computed checksum. | | | |
| 20 | */ | | | |
| 21 | int | | | |
| 22 | pel_id_check(pel) | | | |
| 23 | { | | | |
| 24 | register int checksum; | | | |
| 25 | register unsigned long byte_count; | | | |
| 26 | checksum= ID_CHECKSUM_INIT; | | | |
| 27 | for (byte_count= 0; byte_count < ID_NUM_BYTES; byte_count++) | | | |
| 28 | { | | | |
| 29 | checksum= (checksum << 1) + ((checksum & 0x80) >> 7); | | | |
| 30 | checksum+= (int)pel->idprom.id_prom(byte_count).data; | | | |
| 31 | checksum&= 0xFF; | | | |
| 32 | } | | | |
| 33 | return(checksum); | | | |
| 34 | } | | | |
| 35 | /* | | | |
| 36 | * Load an ID PROM into a character buffer | | | |
| 37 | */ | | | |
| 38 | * address is the physical byte address of the first byte in the ID PROM. | | | |
| 39 | * buffer is the byte address of the first byte in a ID_NUM_BYTES buffer. | | | |
| 40 | * In practice, buffer is really an appropriate ID_PROM_XXX structure, and | | | |
| 41 | * a pointer to it is passed, cast to (u_char *). | | | |
| 42 | */ | | | |
| 43 | void | | | |
| 44 | pel_id_load(pel, buffer) | | | |
| 45 | { | | | |
| 46 | register int byte_count; | | | |
| 47 | for (byte_count= 0; byte_count < ID_NUM_BYTES; byte_count++) | | | |
| 48 | { | | | |
| 49 | *buffer= (unsigned char)pel->idprom.id_prom(byte_count).data; | | | |
| 50 | buffer++; | | | |
| 51 | } | | | |
| 52 | } | | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pel_magic.c

DATE

5/23/89

PAGE #

1/124

TIME

4:41:27 pm

```

1  /* SCCS ID: pel_magic.c rev 3.1.1, 4/24/89 at 07:50:11 */
2
3  .....
4  *
5  *   Pattern play routines
6  *   used in PEL diagnostics
7  *
8  .....
9
10 #include <math.h>
11 #include <common.h>
12 #include <mod_def.h>
13 #include <moduler_extn.h>
14 #include <vrtx.h>
15 #include <lm_diags.h>
16 #include <tmx.h>
17 #include <tmx_extn.h>
18 #include <pac.h>
19 #include <pac_def.h>
20 #include <pac_extn.h>
21 #include <magic.h>
22 #include <pel.h>
23 #include <pel_preamble.h>
24
25 print_magic_register(reg)
26 int reg;
27 {
28     register PEL *pel
29     = (PEL *) (pel_addr(current_lane, current_pel));
30     register long chip;
31
32     for (chip = 4; chip >= 0; chip--) {
33         lm_message("04x ", pel->magic_chip[chip].reg(reg));
34     }
35     lm_message("\n");
36 }
37
38 print_magic_state()
39 {
40     lm_message("DATA OUT (0) = ", print_magic_register(0));
41     lm_message("RDOC (1) = ", print_magic_register(1));
42     lm_message("RDOC (2) = ", print_magic_register(2));
43     lm_message("CTLOUT (3) = ", print_magic_register(3));
44     lm_message("CTLOC (4) = ", print_magic_register(4));
45     lm_message("RDOCN (5) = ", print_magic_register(5));
46     lm_message("DATA IN (6) = ", print_magic_register(6));
47     lm_message("ERROR DATA (7) = ", print_magic_register(7));
48     lm_message("SPORT (8) = ", print_magic_register(8));
49     lm_message("U (9) = ", print_magic_register(9));
50     lm_message("MIE (10) = ", print_magic_register(10));
51     lm_message("VALUE (11) = ", print_magic_register(11));
52     lm_message("TIMING (12) = ", print_magic_register(12));
53     lm_message("STATUS (14) = ", print_magic_register(14));
54 }
55
56 magic_circular_shift()
57 {
58     register PEL *pel
59     = (PEL *) (pel_addr(current_lane, current_pel));
60     register u_short *car = &(pel->car.reg);
61     PAT_WORD pattern;
62
63     *car = 0; /* MUST keep the PEL reset during this test */
64
65     if (!pel_load_pattern_string("z1z1z1z5", 0, STOP_MODE)) {
66         pel_error("Could not load pattern string\n");
67         return (FAILURE);
68     }
69
70     pel_read_pattern(3, &pattern);
71     pattern.pattern.data[0] = pel->magic_chip[4].reg(3);
72     pattern.pattern.data[1] = pel->magic_chip[3].reg(3);
73     pattern.pattern.data[2] = pel->magic_chip[2].reg(3);
74     pattern.pattern.data[3] = pel->magic_chip[1].reg(3);
75     pattern.pattern.data[4] = pel->magic_chip[0].reg(3);
76     pel_write_pattern(3, &pattern);
77
78     if (diag_play() == FAILURE) {
79         pel_error("Could not play pattern\n");
80         pac_play_cleanup();
81         return (FAILURE);
82     }
83     return (SUCCESS);
84 }
85
86 magic_swap()
87 {
88     register PEL *pel
89     = (PEL *) (pel_addr(current_lane, current_pel));
90     register u_short *car = &(pel->car.reg);
91
92     *car = 0; /* MUST keep the PEL reset during this test */
93
94     if (!pel_load_pattern_string("z1z1z1z1", 0, STOP_MODE)) {
95         pel_error("Could not load pattern string\n");
96         return (FAILURE);
97     }
98
99     if (diag_play() == FAILURE) {
100         pel_error("Could not play pattern\n");
101         pac_play_cleanup();
102         return (FAILURE);
103     }
104     return (SUCCESS);
105 }
106
107 char wmc_S1[] = "z1f555d5ASMSCDSE5z5a5u5f1z5a5u50515253545f555d5z1";
108 char wmc_S2[] = "z1z1z1z5";
109 char wmc_S3[] = "z1z1z1z1";
110
111 struct wr_magic_ctl {
112     char *name;
113     char *string;
114     u_short pattern[5];
115 } wr_magic_ctl_data[] = {
116     "CTLOUT", wmc_S1, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
117     "RESET<0>", wmc_S2, 0x5555, 0x5555, 0x5555, 0x5555, 0x5555,
118     "RESET<1>", wmc_S2, 0xcdef, 0x3210, 0x7654, 0xba98, 0xfedc,
119     "SET<0>", wmc_S2, 0x0000, 0xffff, 0xffff, 0xffff, 0xffff,
120     "SET<1>", wmc_S2, 0xffff, 0x0000, 0xffff, 0xffff, 0xffff,

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_magic.c | | DATE 5/23/89 | PAGE # 2/125 |
|--|--|--|--|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | | |
| 121 | | "DOFF" wmc_32, 0xffff, 0xffff, 0x0000, 0xffff, 0xffff, | | | |
| 122 | | "DOIS" wmc_32, 0xffff, 0xffff, 0xffff, 0x0000, 0xffff, | | | |
| 123 | | "LTCMD" wmc_32, 0xffff, 0xffff, 0xffff, 0xffff, 0x0000, | | | |
| 124 | | "LTCMD" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, | | | |
| 125 | | "SDCMD" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, | | | |
| 126 | | "SDCMD" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, | | | |
| 127 | | "FORMAT(0)" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, | | | |
| 128 | | "FORMAT(1)" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, | | | |
| 129 | | "TOGGLE" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, | | | |
| 130 | | "SDLEN(0)" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, | | | |
| 131 | | "SDLEN(1)" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, | | | |
| 132 | | "SDLEN(2)" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, | | | |
| 133 | | "SDLEN(3)" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, | | | |
| 134 | | "SDLEN(0)" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, | | | |
| 135 | | "SDLEN(1)" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, | | | |
| 136 | | "SDLEN(2)" wmc_32, 0x5555, 0x5555, 0x5555, 0x5555, 0x5555, | | | |
| 137 | | "SDLEN(3)" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, | | | |
| 138 | | 0 | | | |
| 139 | | } | | | |
| 140 | | } | | | |
| 141 | | #ifdef info | | | |
| 142 | | string format dodededoddc... where d = data, c = control | | | |
| 143 | | the shift register is x chips by 80 wide (5 chips X 16 bits). | | | |
| 144 | | #endif info | | | |
| 145 | | { | | | |
| 146 | | pel_wr_magic_ctl() | | | |
| 147 | | { | | | |
| 148 | | register PEL *pel | | | |
| 149 | | = (PEL *) (pel_addr(current_lane, current_pel)); | | | |
| 150 | | register u_short *car = &(pel->car.reg); | | | |
| 151 | | register struct wr_magic_ctl *wmc; | | | |
| 152 | | int errors = 0; | | | |
| 153 | | { | | | |
| 154 | | if (diag_clear_errors() != SUCCESS) | | | |
| 155 | | return (FAILURE); | | | |
| 156 | | { | | | |
| 157 | | if (pel_lane_init() != SUCCESS) { | | | |
| 158 | | (void) pel_error("Pia Electronics lane init fails.\n"); | | | |
| 159 | | return FAILURE; | | | |
| 160 | | } | | | |
| 161 | | { | | | |
| 162 | | *car = 0; /* MUST keep the PEL reset during this test */ | | | |
| 163 | | { | | | |
| 164 | | for (wmc = wr_magic_ctl_data, wmc->name, ++wmc) { | | | |
| 165 | | if (pel_play_pattern_string(wmc->string, 0) != SUCCESS) { | | | |
| 166 | | in_error("Lane to Pel to pattern play failed\n", | | | |
| 167 | | current_lane + 'A', current_pel, | | | |
| 168 | | wmc->name); | | | |
| 169 | | return (FAILURE); | | | |
| 170 | | } | | | |
| 171 | | if (compare_magic_chips(3, wmc->pattern) != SUCCESS) { | | | |
| 172 | | in_error("Lane to Pel to compare failed\n", | | | |
| 173 | | current_lane + 'A', current_pel, | | | |
| 174 | | wmc->name); | | | |
| 175 | | ++errors; | | | |
| 176 | | } | | | |
| 177 | | { | | | |
| 178 | | if (errors > 0) | | | |
| 179 | | return (FAILURE); | | | |
| 180 | | else | | | |
| 181 | | return (SUCCESS); | | | |
| 182 | | } | | | |
| 183 | | { | | | |
| 184 | | display_magic_control() | | | |
| 185 | | { | | | |
| 186 | | register struct wr_magic_ctl *wmc; | | | |
| 187 | | register int count; | | | |
| 188 | | { | | | |
| 189 | | setup_good_sample(tmg_measure_period()); | | | |
| 190 | | { | | | |
| 191 | | wmc = wr_magic_ctl_data; | | | |
| 192 | | for (count = 0, count < 11, ++wmc, ++count) { | | | |
| 193 | | in_message("t-10- ", wmc->name); | | | |
| 194 | | print_magic_register(3); | | | |
| 195 | | magic_circular_shift(); | | | |
| 196 | | } | | | |
| 197 | | { | | | |
| 198 | | magic_swap(); | | | |
| 199 | | for (count = 0, count < 11, ++wmc, ++count) { | | | |
| 200 | | in_message("t-10- ", wmc->name); | | | |
| 201 | | print_magic_register(3); | | | |
| 202 | | magic_circular_shift(); | | | |
| 203 | | } | | | |
| 204 | | magic_swap(); | | | |
| 205 | | return (SUCCESS); | | | |
| 206 | | } | | | |
| 207 | | { | | | |
| 208 | | /* compare magic chips(reg, pattern) */ | | | |
| 209 | | /* pattern is a 30 character string representing the 80 bits of expected data */ | | | |
| 210 | | /* "44443333222211110000" */ | | | |
| 211 | | { | | | |
| 212 | | compare_magic_chips(reg, pattern) | | | |
| 213 | | u_short *pattern; | | | |
| 214 | | { | | | |
| 215 | | register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel)); | | | |
| 216 | | char *format = "magic chip %d, Reg %d got %04x, expected %04x\n"; | | | |
| 217 | | register u_short result; | | | |
| 218 | | register long chip_errors; | | | |
| 219 | | u_short errbits; | | | |
| 220 | | int i; | | | |
| 221 | | char buffer[256], buf[8]; | | | |
| 222 | | { | | | |
| 223 | | errors = 0; | | | |
| 224 | | { | | | |
| 225 | | for (chip = 0; chip < 5; ++chip) { | | | |
| 226 | | if ((result = pel_magic_chip(chip).reg[chip]) != pattern[chip]) { | | | |
| 227 | | (void) in_error(format, chip, reg, result, pattern[chip]); | | | |
| 228 | | errors++; | | | |
| 229 | | { | | | |
| 230 | | errbits = result ^ pattern[chip]; | | | |
| 231 | | strcpy(buf, "Check pattern pin(s)"); | | | |
| 232 | | for (i = 0; i < 16; ++i) { | | | |
| 233 | | if (errbits & (1 << i)) { | | | |
| 234 | | sprintf(buf, " %d", chip * 16 + i); | | | |
| 235 | | strcat(buf, " "); | | | |
| 236 | | } | | | |
| 237 | | in_error("%s\n", buf); | | | |
| 238 | | { | | | |
| 239 | | in_error("%s\n", buf); | | | |
| 240 | | } | | | |

1105

5,353,243

1106

| | | | | |
|--|--------------------|-------------------------------------|-----------------|-----------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_magic.c | DATE 5/23/89 | PAGE # 3/126 |
| TIME 4:41:27 pm | | | | |
| LINE # | SOURCE TEXT | | | |
| 241 | } | | | |
| 242 | | | | |
| 243 | if (errors != 0) { | | | |
| 244 | return(FAILURE); | | | |
| 245 | } else { | | | |
| 246 | return(SUCCESS); | | | |
| 247 | } | | | |
| 248 | } | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_opcode.c | DATE 5/23/89 | PAGE # 1/127 |
|--|--|--------------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCSS_ID: pel_opcode.c rev 3.1, 4/24/89 at 07:50:15 */ | | | |
| 2 | /* | | | |
| 3 | * PEL opcode routines | | | |
| 4 | * used in PEL diags. | | | |
| 5 | * | | | |
| 6 | * | | | |
| 7 | * | | | |
| 8 | * | | | |
| 9 | * | | | |
| 10 | #include "common.h" | | | |
| 11 | #include "mod_def.h" | | | |
| 12 | #include "moduler_extn.h" | | | |
| 13 | #include "vtx.h" | | | |
| 14 | #include "lm_diags.h" | | | |
| 15 | #include "tmg.h" | | | |
| 16 | #include "tmg_extn.h" | | | |
| 17 | #include "pac.h" | | | |
| 18 | #include "pac_def.h" | | | |
| 19 | #include "pac_extn.h" | | | |
| 20 | #include "magic.h" | | | |
| 21 | #include "pel.h" | | | |
| 22 | /* | | | |
| 23 | * Define OPCODE_TEST_PERIOD 40000 | | | |
| 24 | * Define EDGE_ON OPCODE_TEST_PERIOD / 4 | | | |
| 25 | * Define EDGE_OFF OPCODE_TEST_PERIOD | | | |
| 26 | * | | | |
| 27 | * | | | |
| 28 | * | | | |
| 29 | * PEL opcode Tests: | | | |
| 30 | * | | | |
| 31 | * pel_patterns_control() | | | |
| 32 | * | | | |
| 33 | * | | | |
| 34 | * register PEL "pel" = (PEL *) (pel_addr(current_lane, current_pel)), | | | |
| 35 | * static long edgetime(s) = { | | | |
| 36 | * EDGE_ON, EDGE_ON, EDGE_ON, EDGE_ON, EDGE_ON, EDGE_ON | | | |
| 37 | * } | | | |
| 38 | * static long period = OPCODE_TEST_PERIOD; | | | |
| 39 | * u_short patterns(5); | | | |
| 40 | * int errors; | | | |
| 41 | * | | | |
| 42 | * if(diag_clear_errors() != SUCCESS) | | | |
| 43 | * return(FAILURE); | | | |
| 44 | * | | | |
| 45 | * pel_reset_all_pels_in_lane(); | | | |
| 46 | * errors = 0; | | | |
| 47 | * | | | |
| 48 | * if (pel_lane_init() != SUCCESS) { /* set up pac/tmg/pel/dab | | | |
| 49 | * (void)lm_error("Unable to initialize lane to\n", current_lane + "A"); | | | |
| 50 | * return(FAILURE); | | | |
| 51 | * } | | | |
| 52 | * pel->car.bit.reset1 = 0; /* disable backplane errors */ | | | |
| 53 | * | | | |
| 54 | * if (tmg_set_timing(period, edgetime, 0L, 100000L, 2500000L) != SUCCESS) { | | | |
| 55 | * return FAILURE; | | | |
| 56 | * } | | | |
| 57 | * | | | |
| 58 | * | | | |
| 59 | * 1. Test opcodes 000 through 110 with PEL enabled and selected for current operation. | | | |
| 60 | * | | | |
| 61 | * lm_message("Testing PEL opcodes 000 through 110 with PEL enabled and selected.\n"); | | | |
| 62 | * if (pel_play_patterns_string("123456789012345678901234567890", 0) != SUCCESS) { | | | |
| 63 | * (void)pel_error("Failure to play OPCODE test patterns"); | | | |
| 64 | * return (FAILURE); | | | |
| 65 | * } | | | |
| 66 | * patterns[4] = patterns[3] = patterns[2] = patterns[1] = patterns[0] = 0xffff; /* check DATA_OUT */ | | | |
| 67 | * if (compare_magic_chips(0, patterns) != SUCCESS) { | | | |
| 68 | * (void)pel_error(" PEL (selected) opcodes 000 through 110 failure \ | | | |
| 69 | * (DATA_OUT)\n"); | | | |
| 70 | * errors++; | | | |
| 71 | * } | | | |
| 72 | * patterns[4] = patterns[3] = patterns[2] = patterns[1] = patterns[0] = 0xaa55; /* check CTL_OUT */ | | | |
| 73 | * if (compare_magic_chips(3, patterns) != SUCCESS) { | | | |
| 74 | * (void)pel_error(" PEL (selected) opcodes 000 through 110 failure \ | | | |
| 75 | * (CTL_OUT)\n"); | | | |
| 76 | * errors++; | | | |
| 77 | * } | | | |
| 78 | * patterns[4] = patterns[3] = patterns[2] = patterns[1] = patterns[0] = 0xffff; /* check BMDEN */ | | | |
| 79 | * if (compare_magic_chips(5, patterns) != SUCCESS) { | | | |
| 80 | * (void)pel_error(" PEL (selected) opcodes 000 through 110 failure \ | | | |
| 81 | * (BMDEN)\n"); | | | |
| 82 | * errors++; | | | |
| 83 | * } | | | |
| 84 | * | | | |
| 85 | * | | | |
| 86 | * | | | |
| 87 | * 2. Test opcode 111 with PEL enabled and selected for current operation. | | | |
| 88 | * | | | |
| 89 | * lm_message("Testing PEL opcode 111 with PEL enabled and selected.\n"); | | | |
| 90 | * if (pel_play_patterns_string("12345678", 0) != SUCCESS) { | | | |
| 91 | * (void)pel_error("Failure to play OPCODE test patterns"); | | | |
| 92 | * return (FAILURE); | | | |
| 93 | * } | | | |
| 94 | * patterns[4] = patterns[3] = patterns[2] = patterns[1] = patterns[0] = 0x0000; /* check DATA_OUT */ | | | |
| 95 | * if (compare_magic_chips(0, patterns) != SUCCESS) { | | | |
| 96 | * (void)pel_error(" PEL (selected) opcode 111 failure (DATA_OUT)\n"); | | | |
| 97 | * errors++; | | | |
| 98 | * } | | | |
| 99 | * patterns[4] = patterns[3] = patterns[2] = patterns[1] = patterns[0] = 0xaa55; /* check CTL_OUT */ | | | |
| 100 | * if (compare_magic_chips(3, patterns) != SUCCESS) { | | | |
| 101 | * (void)pel_error(" PEL (selected) opcode 111 failure (CTL_OUT)\n"); | | | |
| 102 | * errors++; | | | |
| 103 | * } | | | |
| 104 | * patterns[4] = patterns[3] = patterns[2] = patterns[1] = patterns[0] = 0x5555; /* check BMDEN */ | | | |
| 105 | * if (compare_magic_chips(5, patterns) != SUCCESS) { | | | |
| 106 | * (void)pel_error(" PEL (selected) opcode 111 failure (BMDEN)\n"); | | | |
| 107 | * errors++; | | | |
| 108 | * } | | | |
| 109 | * | | | |
| 110 | * | | | |
| 111 | * | | | |
| 112 | * 3. Test opcodes 000 and 001 with PEL enabled and but not selected for current operation. | | | |
| 113 | * | | | |
| 114 | * lm_message("Testing PEL opcodes 000 and 001 with PEL enabled and but not selected.\n"); | | | |
| 115 | * if (pel_play_patterns_string("1234567890", 0) != SUCCESS) { | | | |
| 116 | * (void)pel_error("Failure to play OPCODE test patterns"); | | | |
| 117 | * return (FAILURE); | | | |
| 118 | * } | | | |
| 119 | * patterns[4] = patterns[3] = patterns[2] = patterns[1] = patterns[0] = 0xffff; /* check DATA_OUT */ | | | |
| 120 | * } | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel_opcode.c

DATE 5/23/89 PAGE #
TIME 4:41:28 pm 2/128

```

LINE #          SOURCE TEXT
121  if (compare_magic_chips(0, pattern) != SUCCESS) {
122  (void)pel_error("    PEL (enabled) opcodes 000 and 001 failure \
123  (DATA_OUT)\n");
124  errors++;
125  }
126  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xaaaa; /* check CTL_OUT */
127  if (compare_magic_chips(3, pattern) != SUCCESS) {
128  (void)pel_error("    PEL (enabled) opcodes 000 and 001 failure \
129  (CTL_OUT)\n");
130  errors++;
131  }
132  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x5555; /* check ENDEM */
133  if (compare_magic_chips(5, pattern) != SUCCESS) {
134  (void)pel_error("    PEL (enabled) opcodes 000 and 001 failure \
135  (ENDEM)\n");
136  errors++;
137  }
138  }
139  }
140  }
141  /*
142  * 4. Test opcodes 010 with PEL enabled and but not selected for current operation.
143  */
144  lm_message("Testing PEL opcode 010 with PEL enabled and but not selected.\n");
145  if (pel_play_pattern_string("111227fc", 0) != SUCCESS) {
146  (void)pel_error("Failure to play OPCODE test pattern");
147  return (FAILURE);
148  }
149  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x0000; /* check DATA_OUT */
150  if (compare_magic_chips(0, pattern) != SUCCESS) {
151  (void)pel_error("    PEL (enabled) opcode 010 failure (DATA_OUT)\n");
152  errors++;
153  }
154  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xaaaa; /* check CTL_OUT */
155  if (compare_magic_chips(3, pattern) != SUCCESS) {
156  (void)pel_error("    PEL (enabled) opcode 010 failure (CTL_OUT)\n");
157  errors++;
158  }
159  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x5555; /* check ENDEM */
160  if (compare_magic_chips(5, pattern) != SUCCESS) {
161  (void)pel_error("    PEL (enabled) opcode 010 failure (ENDEM)\n");
162  errors++;
163  }
164  }
165  }
166  /*
167  * 5. Test opcodes 011 and 100 with PEL enabled and but not selected for current operation.
168  */
169  lm_message("Testing PEL opcodes 011 and 100 with PEL enabled and but not selected.\n");
170  if (pel_play_pattern_string("11f422d2", 0) != SUCCESS) {
171  (void)pel_error("Failure to play OPCODE test pattern");
172  return (FAILURE);
173  }
174  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xffff; /* check DATA_OUT */
175  if (compare_magic_chips(0, pattern) != SUCCESS) {
176  (void)pel_error("    PEL (enabled) opcodes 011 and 100 failure \
177  (DATA_OUT)\n");
178  errors++;
179  }
180  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xaaaa; /* check CTL_OUT */
181  if (compare_magic_chips(3, pattern) != SUCCESS) {
182  (void)pel_error("    PEL (enabled) opcodes 011 and 100 failure \
183  (CTL_OUT)\n");
184  errors++;
185  }
186  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x5555; /* check ENDEM */
187  if (compare_magic_chips(5, pattern) != SUCCESS) {
188  (void)pel_error("    PEL (enabled) opcodes 011 and 100 failure \
189  (ENDEM)\n");
190  errors++;
191  }
192  }
193  }
194  /*
195  * 6. Test opcode 101 with PEL enabled and but not selected for current operation.
196  */
197  lm_message("Testing PEL opcode 101 with PEL enabled and but not selected.\n");
198  if (pel_play_pattern_string("11f4227ff", 0) != SUCCESS) {
199  (void)pel_error("Failure to play OPCODE test pattern");
200  return (FAILURE);
201  }
202  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x0000; /* check DATA_OUT */
203  if (compare_magic_chips(0, pattern) != SUCCESS) {
204  (void)pel_error("    PEL (enabled) opcode 101 failure (DATA_OUT)\n");
205  errors++;
206  }
207  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xaaaa; /* check CTL_OUT */
208  if (compare_magic_chips(3, pattern) != SUCCESS) {
209  (void)pel_error("    PEL (enabled) opcode 101 failure (CTL_OUT)\n");
210  errors++;
211  }
212  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x5555; /* check ENDEM */
213  if (compare_magic_chips(5, pattern) != SUCCESS) {
214  (void)pel_error("    PEL (enabled) opcode 101 failure (ENDEM)\n");
215  errors++;
216  }
217  }
218  }
219  /*
220  * 7. Test opcodes 110 and 111 with PEL enabled and but not selected for current operation.
221  */
222  lm_message("Testing PEL opcodes 110 and 111 with PEL enabled and but not selected.\n");
223  if (pel_play_pattern_string("11f422b2c22f22b2", 0) != SUCCESS) {
224  (void)pel_error("Failure to play OPCODE test pattern");
225  return (FAILURE);
226  }
227  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xffff; /* check DATA_OUT */
228  if (compare_magic_chips(0, pattern) != SUCCESS) {
229  (void)pel_error("    PEL (enabled) opcodes 110 and 111 failure \
230  (DATA_OUT)\n");
231  errors++;
232  }
233  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xaaaa; /* check CTL_OUT */
234  if (compare_magic_chips(3, pattern) != SUCCESS) {
235  (void)pel_error("    PEL (enabled) opcodes 110 and 111 failure \
236  (CTL_OUT)\n");
237  errors++;
238  }
239  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x5555; /* check ENDEM */
240  if (compare_magic_chips(5, pattern) != SUCCESS) {
241  (void)pel_error("    PEL (enabled) opcodes 110 and 111 failure \

```

1111

5,353,243

1112

| | | | | |
|--|------------------------------------|--------------------------------------|-----------------|-----------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_opcode.c | DATE 5/23/89 | PAGE # 3/129 |
| TIME 4:41:28 pm | | | | |
| LINE # | SOURCE TEXT | | | |
| 241 | (BNDEN*)\n"); | | | |
| 242 | errors++; | | | |
| 243 | } | | | |
| 244 | | | | |
| 245 | return errors ? FAILURE : SUCCESS; | | | |
| 246 | } | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_pattern.c | DATE 5/23/89 | PAGE # 1/130 |
|--|---|---------------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCCS ID: pel_pattern.c Rev 3.1, 4/24/89 at 07:58:13 */ | | | |
| 2 | /* | | | |
| 3 | Pattern play routines | | | |
| 4 | used in PEL diagnostics | | | |
| 5 | */ | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| 10 | #include <math.h> | | | |
| 11 | #include "common.h" | | | |
| 12 | #include "mod_def.h" | | | |
| 13 | #include "modeler_extn.h" | | | |
| 14 | #include "vtx.h" | | | |
| 15 | #include "lm_diags.h" | | | |
| 16 | #include "tag.h" | | | |
| 17 | #include "tag_extn.h" | | | |
| 18 | #include "pac.h" | | | |
| 19 | #include "pac_def.h" | | | |
| 20 | #include "pac_extn.h" | | | |
| 21 | #include "magic.h" | | | |
| 22 | #include "pel.h" | | | |
| 23 | #include "pel_preamble.h" | | | |
| 24 | | | | |
| 25 | | | | |
| 26 | | | | |
| 27 | | | | |
| 28 | | | | |
| 29 | | | | |
| 30 | | | | |
| 31 | | | | |
| 32 | | | | |
| 33 | | | | |
| 34 | | | | |
| 35 | | | | |
| 36 | | | | |
| 37 | | | | |
| 38 | | | | |
| 39 | | | | |
| 40 | | | | |
| 41 | | | | |
| 42 | | | | |
| 43 | | | | |
| 44 | | | | |
| 45 | | | | |
| 46 | | | | |
| 47 | | | | |
| 48 | | | | |
| 49 | | | | |
| 50 | | | | |
| 51 | | | | |
| 52 | | | | |
| 53 | | | | |
| 54 | | | | |
| 55 | | | | |
| 56 | | | | |
| 57 | | | | |
| 58 | | | | |
| 59 | | | | |
| 60 | | | | |
| 61 | | | | |
| 62 | | | | |
| 63 | | | | |
| 64 | | | | |
| 65 | | | | |
| 66 | | | | |
| 67 | | | | |
| 68 | | | | |
| 69 | | | | |
| 70 | | | | |
| 71 | | | | |
| 72 | | | | |
| 73 | | | | |
| 74 | | | | |
| 75 | | | | |
| 76 | | | | |
| 77 | | | | |
| 78 | | | | |
| 79 | | | | |
| 80 | | | | |
| 81 | | | | |
| 82 | | | | |
| 83 | | | | |
| 84 | | | | |
| 85 | | | | |
| 86 | | | | |
| 87 | | | | |
| 88 | | | | |
| 89 | | | | |
| 90 | | | | |
| 91 | | | | |
| 92 | | | | |
| 93 | | | | |
| 94 | | | | |
| 95 | | | | |
| 96 | | | | |
| 97 | | | | |
| 98 | | | | |
| 99 | | | | |
| 100 | | | | |
| 101 | | | | |
| 102 | | | | |
| 103 | | | | |
| 104 | | | | |
| 105 | | | | |
| 106 | | | | |
| 107 | | | | |
| 108 | | | | |
| 109 | | | | |
| 110 | | | | |
| 111 | | | | |
| 112 | | | | |
| 113 | | | | |
| 114 | | | | |
| 115 | | | | |
| 116 | | | | |
| 117 | | | | |
| 118 | | | | |
| 119 | | | | |
| 120 | | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_pattern.c | DATE 5/13/89 | PAGE # 2/131 |
|--|--|---|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 121 | | "Display pattern memory preamble (short)", | | |
| 122 | | display_pattern_preamble, | | |
| 123 | | LM_DIAG_utility, | | |
| 124 | | LM_DIAG_mail, | | |
| 125 | | 0 | | |
| 126 | | { | | |
| 127 | | } | | |
| 128 | | "4", | | |
| 129 | | "Display pattern memory preamble (long)", | | |
| 130 | | display_pattern_preamble_verbosely, | | |
| 131 | | LM_DIAG_utility, | | |
| 132 | | LM_DIAG_mail, | | |
| 133 | | 0 | | |
| 134 | | { | | |
| 135 | | } | | |
| 136 | | "5", | | |
| 137 | | "Display pattern string code", | | |
| 138 | | display_pattern_string_code, | | |
| 139 | | LM_DIAG_utility, | | |
| 140 | | LM_DIAG_mail, | | |
| 141 | | 0 | | |
| 142 | | { | | |
| 143 | | } | | |
| 144 | | "6", | | |
| 145 | | "Build Preamble", | | |
| 146 | | build_preamble, | | |
| 147 | | LM_DIAG_utility, | | |
| 148 | | LM_DIAG_mail, | | |
| 149 | | 0 | | |
| 150 | | { | | |
| 151 | | } | | |
| 152 | | static LM_DIAG_MENU menu = | | |
| 153 | | { | | |
| 154 | | "PEL UTILITIES", | | |
| 155 | | sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM), | | |
| 156 | | 0, | | |
| 157 | | menu_list | | |
| 158 | | } | | |
| 159 | | menu.title = parent_menu-> | | |
| 160 | | menu.item[parent_menu->current_selection].menu_text, | | |
| 161 | | return lm_display_menu(menu); | | |
| 162 | | } | | |
| 163 | | } | | |
| 164 | | #define MIN_PATTERN_LENGTH 4 | | |
| 165 | | #define ODD_ERROR -1 | | |
| 166 | | pel_pattern_count(pattern_string) | | |
| 167 | | char *pattern_string; | | |
| 168 | | { | | |
| 169 | | register long len; | | |
| 170 | | static char odd_error[] = | | |
| 171 | | "Pattern string has odd number of chars:\n\t\t\"%s\\n\\n", | | |
| 172 | | { | | |
| 173 | | if ((len = strlen(pattern_string)) % 2) { | | |
| 174 | | (void)lm_error(odd_error, pattern_string); | | |
| 175 | | return ODD_ERROR; | | |
| 176 | | } | | |
| 177 | | return len >> 1; /* pattern count */ | | |
| 178 | | } | | |
| 179 | | } | | |
| 180 | | build_pattern_data(pattern_string, first_pattern_no) | | |
| 181 | | char *pattern_string; | | |
| 182 | | { | | |
| 183 | | struct string_pattern *pel_decode_string_patterns(), | | |
| 184 | | static char set_error[] = | | |
| 185 | | "Pattern memory set error at line = %d, address = %08x\\n", | | |
| 186 | | { | | |
| 187 | | PAT_WORD pattern; | | |
| 188 | | register long i; | | |
| 189 | | register u_long a, pattern_count; | | |
| 190 | | struct string_pattern *string_pattern; | | |
| 191 | | { | | |
| 192 | | if ((pattern_count = pel_pattern_count(pattern_string)) == ODD_ERROR) { | | |
| 193 | | return (0); | | |
| 194 | | } | | |
| 195 | | for (a = first_pattern_no; pattern_string != '\\0'; ++a) { | | |
| 196 | | /* set the pattern data according to first character in pair */ | | |
| 197 | | if (!((string_pattern | | |
| 198 | | = pel_decode_string_pattern(pattern_string))) { | | |
| 199 | | lm_message("Bad pel_load_pattern_string\\n"); | | |
| 200 | | return(0); | | |
| 201 | | } | | |
| 202 | | for (i = 0; i < 5; i++) | | |
| 203 | | pattern.pattern.data[i] = string_pattern->data[i]; | | |
| 204 | | pattern_string++; /* move to second pair of characters */ | | |
| 205 | | /* compute pel_control and pel_address | | |
| 206 | | from second of character pair */ | | |
| 207 | | if ((pattern_string > '\\0') && (pattern_string < '\\7')) { | | |
| 208 | | pattern.pattern.pel_control | | |
| 209 | | = pattern_string - '\\0'; | | |
| 210 | | pattern.pattern.pel_address = current_pel; | | |
| 211 | | } else if ((pattern_string > '\\a') && (pattern_string < '\\h')) { | | |
| 212 | | pattern.pattern.pel_control | | |
| 213 | | = pattern_string - '\\a'; | | |
| 214 | | pattern.pattern.pel_address | | |
| 215 | | = (current_pel + 1) & 16; | | |
| 216 | | } else { | | |
| 217 | | lm_message("Bad pel_load_pattern_string\\n"); | | |
| 218 | | return(0); | | |
| 219 | | } | | |
| 220 | | /* Make sure the branch bits and the stop bit are zero (NOP) */ | | |
| 221 | | pattern.pattern.branch = 0; | | |
| 222 | | pattern.pattern.stop = 0; | | |
| 223 | | /* Set the spare bits to zero so that pattern loading is */ | | |
| 224 | | consistent. Clear the USER bit */ | | |
| 225 | | pattern.pattern.backplane_spare = 0; | | |
| 226 | | pattern.pattern.pcc_spare = 0; | | |
| 227 | | pattern.pattern.user = 0; | | |
| 228 | | pattern_string++; /* move to next pair of characters */ | | |
| 229 | | /* write the pattern to pattern memory, | | |
| 230 | | and move to next address */ | | |
| 231 | | if (pel_write_pattern(a, &pattern) != SUCCESS) { | | |
| 232 | | } | | |
| 233 | | } | | |
| 234 | | } | | |
| 235 | | } | | |
| 236 | | } | | |
| 237 | | } | | |
| 238 | | } | | |
| 239 | | } | | |
| 240 | | } | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel_pattern.c

DATE 5/23/89
TIME 4:41:28 pm

PAGE #
3/132

```

LINE #          SOURCE TEXT
241      ln_message(set_error, current_lane, n);
242      return(0);
243
244      }
245      return (pattern_count);
246
247  }
248  pel_type_pattern_string()
249  {
250      char pattern[256];
251      long start_pattern = 0, dab_type = 0;
252
253      pattern[0] = '\0';
254      ln_get_input("Enter pattern string: ", pattern, 256);
255      diag_get_long(&start_pattern, "start pattern number", 01,
256      (long)pac(current_lane).num_patterns);
257      diag_get_long(&dab_type,
258      "Enter DAB type (Public_DAB = 0, PRIVATE_DAB = 1)", 01, 11);
259
260      if (set_vlog1(0.8) == FAILURE) return(FAILURE);
261      if (set_vlog2(2.0) == FAILURE) return(FAILURE);
262      if (set_vth1(0.3) == FAILURE) return(FAILURE);
263      if (set_vth2(0.4) == FAILURE) return(FAILURE);
264      if (set_vth3(4.0) == FAILURE) return(FAILURE);
265      if (set_vth4(4.4) == FAILURE) return(FAILURE);
266
267      if (pel_lane_init() != SUCCESS) { /* set up pac/seg/pel/dab */
268          (void)ln_error("Unable to initialize lane to %c", current_lane + 'A');
269          return(FAILURE);
270      }
271      if (pel_dab_init(dab_type) != SUCCESS) {
272          (void)ln_error("Unable to initialize DAB to %c, PEL id %c",
273          current_lane + 'A', current_pel);
274          return(FAILURE);
275      }
276
277      if (pel_play_pattern_string(pattern, start_pattern) == SUCCESS) {
278          print_magic_state();
279          return SUCCESS;
280      }
281      ln_message("ERROR playing pattern string\n");
282      return FAILURE;
283
284  }
285
286  pel_enter_pattern_string()
287  {
288      char pattern[256];
289      long start_pattern = 0;
290
291      pattern[0] = '\0';
292      ln_get_input("Enter pattern string: ", pattern, 256);
293      diag_get_long(&start_pattern, "start pattern number", 01,
294      (long)pac(current_lane).num_patterns);
295
296      if (!pel_load_pattern_string(pattern, start_pattern, STOP_MODE)) {
297          (void)pel_error("pel_load_pattern_string failed\n");
298          return (FAILURE);
299      }
300      return (SUCCESS);
301
302  }
303
304  pel_play_pattern_string(pattern_string, first_pattern_no)
305  char *pattern_string,
306  {
307      if (!pel_load_pattern_string(pattern_string,
308      first_pattern_no, STOP_MODE)) {
309          (void)pel_error("pel_play_pattern_string failed\n");
310          return (FAILURE);
311      }
312
313      if (diag_play() == FAILURE) {
314          (void)pel_error("pel_play_pattern_string failed in diag_play\n");
315          return(FAILURE);
316      }
317      return (SUCCESS);
318  }
319
320  pel_load_pattern_string(pattern_string, first_pattern_no, mode)
321  char *pattern_string,
322  {
323      register u_long pattern_count;
324
325      if ((pattern_count = build_pattern_data(pattern_string,
326      first_pattern_no)) == 0) {
327          (void)pel_error("Can't load pattern data\n");
328          return 0;
329      }
330
331      if (pel_build_pattern_control(first_pattern_no, pattern_count, mode)
332      != SUCCESS) {
333          (void)pel_error("Can't load pattern control\n");
334          return(0);
335      }
336
337      return pattern_count;
338  }
339
340  struct string_pattern *
341  pel_decode_string_patterns(c)
342  char c;
343  {
344      struct string_pattern *p;
345
346      for (p = String_patterns; p->code; ++p) {
347          if (c == p->code) return p;
348      }
349      return 0; /* undefined code */
350  }
351
352  char
353  pel_encode_string_patterns(d)
354  short d[];
355  {
356      register int chip;
357      register struct string_pattern *p;
358
359      for (p = String_patterns; p->code; ++p) {
360          for (chip = 0; chip < 5; ++chip) {

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_pattern.c | DATE 5/23/89 | PAGE # 4/133 |
|--|--|--|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 361 | | if (p->data[chip] != d[chip]) break; | | |
| 362 | | if (chip == 5) return p->code; | | |
| 363 | | return 0; /* undefined code */ | | |
| 364 | | } | | |
| 365 | | display_pattern_preamble() | | |
| 366 | | { | | |
| 367 | | pattern_preamble(0); | | |
| 368 | | } | | |
| 369 | | display_pattern_preamble_verbose() | | |
| 370 | | { | | |
| 371 | | pattern_preamble(1); | | |
| 372 | | } | | |
| 373 | | set_pattern_preamble(start_pattern, preamble_array) | | |
| 374 | | { | | |
| 375 | | u_long start_pattern; | | |
| 376 | | preamble preamble_array[]; | | |
| 377 | | { | | |
| 378 | | PAT_WORD patptr; | | |
| 379 | | u_long pattern, limit_pattern = start_pattern + PREAMBLE_SIZE; | | |
| 380 | | u_long pin, start_pin, limit_pin; | | |
| 381 | | u_long chip; | | |
| 382 | | for (pin = 0; pin < 80; ++pin) { | | |
| 383 | | preamble_array[pin].x = 0; | | |
| 384 | | } | | |
| 385 | | for (pattern = start_pattern; pattern < limit_pattern; ++pattern) { | | |
| 386 | | pel_read_pattern(pattern, &patptr); | | |
| 387 | | for (chip = 0; chip < 5; ++chip) { | | |
| 388 | | start_pin = (4 - chip) * 16; | | |
| 389 | | limit_pin = start_pin + 16; | | |
| 390 | | for (pin = start_pin; pin < limit_pin; ++pin) { | | |
| 391 | | if ((patptr.pattern.data[chip] & (1 << (pin & 16))) != 0) | | |
| 392 | | preamble_array[pin].x = | | |
| 393 | | (1 << (pattern - start_pattern)); | | |
| 394 | | } | | |
| 395 | | } | | |
| 396 | | return SUCCESS; | | |
| 397 | | } | | |
| 398 | | set_pattern_preamble(start_pattern, preamble_array) | | |
| 399 | | { | | |
| 400 | | u_long start_pattern; | | |
| 401 | | preamble preamble_array[]; | | |
| 402 | | { | | |
| 403 | | PAT_WORD patptr; | | |
| 404 | | u_long pattern, limit_pattern; | | |
| 405 | | u_long pin, start_pin, limit_pin; | | |
| 406 | | u_long chip; | | |
| 407 | | limit_pattern = start_pattern + PREAMBLE_SIZE; | | |
| 408 | | for (pattern = start_pattern; pattern < limit_pattern; ++pattern) { | | |
| 409 | | pel_read_pattern(pattern, &patptr); | | |
| 410 | | for (chip = 0; chip < 5; ++chip) { | | |
| 411 | | patptr.pattern.data[chip] = 0; | | |
| 412 | | start_pin = (4 - chip) * 16; | | |
| 413 | | limit_pin = start_pin + 16; | | |
| 414 | | for (pin = start_pin; pin < limit_pin; ++pin) { | | |
| 415 | | if ((patptr.pattern.data[chip] & (1 << (pin & 16))) != 0) | | |
| 416 | | preamble_array[pin].x = | | |
| 417 | | (1 << (pattern - start_pattern)); | | |
| 418 | | } | | |
| 419 | | if ((preamble_array[pin].x & (1 << (pattern - start_pattern))) != 0) | | |
| 420 | | patptr.pattern.data[chip] = (1 << (pin & 16)); | | |
| 421 | | } | | |
| 422 | | } | | |
| 423 | | pel_write_pattern(pattern, &patptr); | | |
| 424 | | } | | |
| 425 | | return SUCCESS; | | |
| 426 | | } | | |
| 427 | | pattern_preamble(verbose) | | |
| 428 | | { | | |
| 429 | | register int count; | | |
| 430 | | preamble preamble_array[80]; | | |
| 431 | | u_long start_pattern; | | |
| 432 | | int pin; | | |
| 433 | | int chip; | | |
| 434 | | start_pattern = 0; | | |
| 435 | | diag_get_long(&start_pattern, "start pattern", start_pattern, | | |
| 436 | | (u_long)(pac(current_lane).num_patterns - 1)); | | |
| 437 | | set_pattern_preamble(start_pattern, preamble_array); | | |
| 438 | | { | | |
| 439 | | if (verbose) { | | |
| 440 | | for (count = pin = 0; pin < 80; ++pin, ++count) { | | |
| 441 | | if (count % 20) { | | |
| 442 | | if (!count - more(pin, 80, count)) | | |
| 443 | | break; | | |
| 444 | | lm_message("pin %2d", pin); | | |
| 445 | | decode_preamble(preamble_array[pin].x); | | |
| 446 | | } | | |
| 447 | | } else { | | |
| 448 | | for (pin = 0; pin < 16; ++pin) { | | |
| 449 | | lm_message("pin %2d", pin); | | |
| 450 | | for (chip = 0; chip < 5; ++chip) { | | |
| 451 | | lm_message("pin %2d %07x", | | |
| 452 | | pin + 16 * chip, preamble_array[pin + 16 * chip].x); | | |
| 453 | | } | | |
| 454 | | lm_message("\n"); | | |
| 455 | | } | | |
| 456 | | return SUCCESS; | | |
| 457 | | } | | |
| 458 | | static char *array_format[] = | | |
| 459 | | { | | |
| 460 | | "RC " | | |
| 461 | | "RZ " | | |
| 462 | | "R1 " | | |
| 463 | | "DNRL" | | |
| 464 | | }; | | |
| 465 | | static char *array_toggle[] = | | |
| 466 | | { | | |
| 467 | | " | | |
| 468 | | " | | |
| 469 | | " | | |
| 470 | | " | | |
| 471 | | " | | |
| 472 | | " | | |
| 473 | | " | | |
| 474 | | " | | |
| 475 | | " | | |
| 476 | | " | | |
| 477 | | " | | |
| 478 | | " | | |
| 479 | | " | | |
| 480 | | " | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_pattern.c | DATE 5/23/89 | PAGE # 5/134 |
|--|---|---------------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 481 | /* TOG */ | | | |
| 482 | { | | | |
| 483 | static char *array_doff[] = | | | |
| 484 | { | | | |
| 485 | /* DOFF0 */ | | | |
| 486 | /* DOFF5 */ | | | |
| 487 | } | | | |
| 488 | static char *array_hddis[] = | | | |
| 489 | { | | | |
| 490 | /* HDDIS */ | | | |
| 491 | } | | | |
| 492 | static char *array_lcycmd[] = | | | |
| 493 | { | | | |
| 494 | /* LCM */ | | | |
| 495 | /* LCM */ | | | |
| 496 | } | | | |
| 497 | static char *array_lcychn[] = | | | |
| 498 | { | | | |
| 499 | /* LCM */ | | | |
| 500 | /* LCM */ | | | |
| 501 | } | | | |
| 502 | static char *array_seqmd[] = | | | |
| 503 | { | | | |
| 504 | /* SEQMD */ | | | |
| 505 | /* SEQMD */ | | | |
| 506 | } | | | |
| 507 | static char *array_seqchn[] = | | | |
| 508 | { | | | |
| 509 | /* SEQMD */ | | | |
| 510 | /* SEQMD */ | | | |
| 511 | } | | | |
| 512 | static char *array_hmden[] = | | | |
| 513 | { | | | |
| 514 | /* HMDEN */ | | | |
| 515 | /* HMDEN */ | | | |
| 516 | } | | | |
| 517 | } | | | |
| 518 | } | | | |
| 519 | decode_preamble(x) | | | |
| 520 | { | | | |
| 521 | preamble x; | | | |
| 522 | { | | | |
| 523 | lm_message("to to SLEWx XENx Rtd Std to to to to to to to\n", | | | |
| 524 | array_format(x.b.format), | | | |
| 525 | array_toggle(x.b.toggle), | | | |
| 526 | x.b.adlen, | | | |
| 527 | x.b.adhen, | | | |
| 528 | x.b.reset, | | | |
| 529 | x.b.set + 2, | | | |
| 530 | array_doff(x.b.doff), | | | |
| 531 | array_hddis(x.b.hddis), | | | |
| 532 | array_lcycmd(x.b.lcycmd), | | | |
| 533 | array_lcychn(x.b.lcychn), | | | |
| 534 | array_seqmd(x.b.seqmd), | | | |
| 535 | array_seqchn(x.b.seqchn), | | | |
| 536 | array_hmden(x.b.hmden)); | | | |
| 537 | } | | | |
| 538 | } | | | |
| 539 | build_preamble() | | | |
| 540 | { | | | |
| 541 | char reply(DIAG_MAX_INPUT); | | | |
| 542 | preamble preamble_array[80]; | | | |
| 543 | preamble x; | | | |
| 544 | ulong start_patterns, | | | |
| 545 | ulong pin, start_pin, end_pin, | | | |
| 546 | ulong format, | | | |
| 547 | ulong toggle, | | | |
| 548 | ulong adlen, | | | |
| 549 | ulong reset, | | | |
| 550 | ulong set, | | | |
| 551 | ulong doff, | | | |
| 552 | ulong hddis, | | | |
| 553 | ulong lcycmd, | | | |
| 554 | ulong lcychn, | | | |
| 555 | ulong seqmd, | | | |
| 556 | ulong seqchn, | | | |
| 557 | ulong hmden, | | | |
| 558 | char *get_range(); | | | |
| 559 | { | | | |
| 560 | start_patterns = 0; | | | |
| 561 | diag_get_ulong(&start_patterns, "start patterns", start_patterns, | | | |
| 562 | (ulong)((pc(current_line).num_patterns - 1)); | | | |
| 563 | { | | | |
| 564 | lm_get_input("Enter pin list (P P.P. P-P....) "); | | | |
| 565 | reply, (short)DIAG_MAX_INPUT); | | | |
| 566 | { | | | |
| 567 | if (!((s = get_range(reply, &start_pin, &end_pin))) { | | | |
| 568 | lm_message("Although s = td and e = td\n", start_pin, end_pin); | | | |
| 569 | (void)pel_error("Illegal pin selection string\n"); | | | |
| 570 | return FAILURE; | | | |
| 571 | } | | | |
| 572 | { | | | |
| 573 | if ((start_pin < 0) (start_pin > 79)) { | | | |
| 574 | (void)lm_error("Illegal pin number td\n", start_pin); | | | |
| 575 | return FAILURE; | | | |
| 576 | } | | | |
| 577 | { | | | |
| 578 | get_patterns_preamble(start_patterns, preamble_array); | | | |
| 579 | x.x = preamble_array[start_pin].x; | | | |
| 580 | format = x.b.format; | | | |
| 581 | toggle = x.b.toggle; | | | |
| 582 | adlen = (x.b.adlen) & 0xf; | | | |
| 583 | adhen = x.b.adhen; | | | |
| 584 | reset = x.b.reset; | | | |
| 585 | set = x.b.set + 2; | | | |
| 586 | doff = x.b.doff; | | | |
| 587 | hddis = (x.b.hddis) & 0xf; | | | |
| 588 | lcycmd = (x.b.lcycmd) & 0xf; | | | |
| 589 | lcychn = (x.b.lcychn) & 0xf; | | | |
| 590 | seqmd = (x.b.seqmd) & 0xf; | | | |
| 591 | seqchn = (x.b.seqchn) & 0xf; | | | |
| 592 | hmden = (x.b.hmden) & 0xf; | | | |
| 593 | { | | | |
| 594 | diag_get_ulong(&format, "data format: 0->RC, 1->RZ, 2->R1, 3->DNRZ", 01, 31); | | | |
| 595 | diag_get_ulong(&toggle, "toggle", 01, 11); | | | |
| 596 | diag_get_ulong(&adlen, "adlen", 01, 151); | | | |
| 597 | diag_get_ulong(&adhen, "adhen", 01, 151); | | | |
| 598 | diag_get_ulong(&reset, "reset edge", 01, 31); | | | |
| 599 | diag_get_ulong(&set, "set edge", 21, 51); | | | |
| 600 | diag_get_ulong(&doff, "driver off edge: 0->0, 1->5", 01, 11); | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_pattern.c | DATE 5/23/89 | PAGE # 6/135 |
|--|--|---|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 601 | | diag_get_ulong(&hddia, "hddia", 01, 11); | | |
| 602 | | diag_get_ulong(&lcycmd, "lcycmd", 01, 11); | | |
| 603 | | diag_get_ulong(&lcychd, "lcychd", 01, 11); | | |
| 604 | | diag_get_ulong(&seqmd, "seqmd", 01, 11); | | |
| 605 | | diag_get_ulong(&seqhd, "seqhd", 01, 11); | | |
| 606 | | diag_get_ulong(&hmden, "hmden", 01, 11); | | |
| 607 | | x.b.format = format; | | |
| 608 | | x.b.toggle = toggle; | | |
| 609 | | x.b.adlen = adlen; | | |
| 610 | | x.b.adhen = adhen; | | |
| 611 | | x.b.reset = reset; | | |
| 612 | | x.b.set = set - 2; | | |
| 613 | | x.b.doff = doff; | | |
| 614 | | x.b.hddia = hddia; | | |
| 615 | | x.b.lcycmd = lcycmd; | | |
| 616 | | x.b.lcychd = lcychd; | | |
| 617 | | x.b.seqmd = seqmd; | | |
| 618 | | x.b.seqhd = seqhd; | | |
| 619 | | x.b.hmden = hmden; | | |
| 620 | | lm_message("%X\n", x.x); | | |
| 621 | | do { | | |
| 622 | | lm_message("td ... td\n", start_pin, end_pin); | | |
| 623 | | if ((start_pin < 0) (start_pin > 79)) { | | |
| 624 | | (void)lm_error("Illegal pin number %d\n", start_pin); | | |
| 625 | | return FAILURE; | | |
| 626 | | } | | |
| 627 | | if ((end_pin < 0) (end_pin > 79)) { | | |
| 628 | | (void)lm_error("Illegal pin number %d\n", end_pin); | | |
| 629 | | return FAILURE; | | |
| 630 | | } | | |
| 631 | | for (pin = start_pin; pin <= end_pin; ++pin) { | | |
| 632 | | preamble_array[pin].x = x.x; | | |
| 633 | | } | | |
| 634 | | while (a = get_range(a, start_pin, end_pin)); | | |
| 635 | | set_patterns_preamble(start_patterns, preamble_array); | | |
| 636 | | return SUCCESS; | | |
| 637 | | } | | |
| 638 | | char *preamble_bit_meaning[] = { | | |
| 639 | | "start preamble", | | |
| 640 | | "format", | | |
| 641 | | "toggle", | | |
| 642 | | "adlen", | | |
| 643 | | "adhen", | | |
| 644 | | "reset", | | |
| 645 | | "set", | | |
| 646 | | "doff", | | |
| 647 | | "hddia", | | |
| 648 | | "lcycmd", | | |
| 649 | | "lcychd", | | |
| 650 | | "seqmd", | | |
| 651 | | "seqhd", | | |
| 652 | | "hmden", | | |
| 653 | | "end preamble", | | |
| 654 | | }; | | |
| 655 | | print_preamble_line_meaning(bit) | | |
| 656 | | { | | |
| 657 | | if ((bit >= 0) && (bit < PREAMBLE_SIZE)) { | | |
| 658 | | lm_message(preamble_bit_meaning[bit]); | | |
| 659 | | } | | |
| 660 | | display_pattern_string_code() | | |
| 661 | | { | | |
| 662 | | char buffer[80]; | | |
| 663 | | long start_patterns, end_patterns, this_end; | | |
| 664 | | int count, origia; | | |
| 665 | | start_patterns = 0; | | |
| 666 | | end_patterns = pac[current_lane].num_patterns - 1; | | |
| 667 | | diag_get_long(&start_patterns, "start patterns", start_patterns, end_patterns); | | |
| 668 | | diag_get_long(&end_patterns, "end patterns", start_patterns, end_patterns); | | |
| 669 | | count = 0; | | |
| 670 | | origia = start_patterns; | | |
| 671 | | for (this_end = start_patterns + 1; start_patterns < end_patterns; | | |
| 672 | | start_patterns += 32; this_end += 32) { | | |
| 673 | | if (this_end > end_patterns) this_end = end_patterns; | | |
| 674 | | lm_message("%10d: ", start_patterns); | | |
| 675 | | pel_get_patterns_string(start_patterns, this_end, buffer); | | |
| 676 | | lm_message("%s\n", buffer); | | |
| 677 | | if ((start_patterns < end_patterns) && !(++count % 20)) { | | |
| 678 | | if (!(count = more((int)(start_patterns - origia), | | |
| 679 | | (end_patterns - origia), count))) | | |
| 680 | | break; | | |
| 681 | | } | | |
| 682 | | return SUCCESS; | | |
| 683 | | } | | |
| 684 | | pel_get_patterns_string(start_patterns, end_patterns, s) | | |
| 685 | | u_long start_patterns, end_patterns; | | |
| 686 | | char *s; | | |
| 687 | | { | | |
| 688 | | PAT_WORD patptr; | | |
| 689 | | u_long patterns; | | |
| 690 | | for (patterns = start_patterns; patterns <= end_patterns; ++patterns) { | | |
| 691 | | pel_read_patterns(patterns, &patptr); | | |
| 692 | | if (!(*s = pel_encode_string_patterns(patptr.pattern.data))) | | |
| 693 | | *s = "?"; | | |
| 694 | | } | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel_pattern.c

DATE
5/23/89
TIME
4:41:28 pm

PAGE #
7/136

LINE # SOURCE TEXT

```
721 ++;
722 *g++ = '0' + patptr->pattern.pel_control;
723 }
724 *s = '\0';
725 return SUCCESS;
726 }
727
728 display_pattern_memory()
729 {
730     register int i, count;
731     FAT_WORD patptr;
732     long start_pattern, stop_pattern, start_preamble;
733     char pattern_string[3];
734
735     start_pattern = 0;
736     stop_pattern = pec[current_lease].num_patterns - 1;
737     diag_get_long(&start_pattern, "start pattern", start_pattern, stop_pattern);
738     diag_get_long(&stop_pattern, "stop pattern", start_pattern, stop_pattern);
739
740     start_preamble = stop_pattern + 1; /* don't display preamble bits
741                                         if no preamble */
742     for (i = start_pattern, count = 0; i <= stop_pattern; ++i) {
743         pel_read_pattern(i, patptr);
744         lm_message("10d: 1041 1042 1043 1044 1045 1046 ",
745                 i,
746                 patptr->pattern.data[0],
747                 patptr->pattern.data[1],
748                 patptr->pattern.data[2],
749                 patptr->pattern.data[3],
750                 patptr->pattern.data[4]);
751         lm_message(" PAtt2d", patptr->pattern.pel_address);
752         lm_message(" PCd", patptr->pattern.pel_control);
753         switch (patptr->pattern.branch) {
754             case 1: lm_message(" BA"); break;
755             case 2: lm_message(" BF"); break;
756             case 3: lm_message(" BK"); break;
757             default: lm_message(" "); break;
758         }
759         lm_message("33s ", (patptr->pattern.stop)? "STOP": "");
760         lm_message("33s ", (patptr->pattern.user)? "USER": "");
761         pel_get_pattern_string(i, 1, pattern_string);
762         lm_message("3s ", pattern_string);
763
764         if (patptr->pattern.pel_control == 2)
765             start_preamble = i; /* start of preamble */
766         print_preamble_line_meaning(i - start_preamble);
767         lm_message("\n");
768         if ((i < stop_pattern) && (++count % 20) {
769             if (!(count == more(i - (1st)start_pattern,
770                                 stop_pattern - start_pattern, count)))
771                 break;
772         }
773     }
774 }
775
776 pel_read_pattern(pattern, patptr)
777 int pattern;
778 FAT_WORD *patptr;
779 {
780     extern u_long *Host_memory;
781     register u_long i, *mem;
782
783     if (!Host) return pec_read_pattern(pattern, patptr);
784
785     /* Check if pattern number is valid */
786     if((pattern < 0) || (pattern > 0x40000))
787         return(FAILURE);
788
789     mem = 4(Host_memory[4 * pattern]);
790
791     /* Read pattern word */
792     for (i = 0; i < 3; ++i)
793         patptr->mem_bank[i] = *mem++;
794     return(SUCCESS);
795 }
796
797 pel_write_pattern(pattern, patptr)
798 int pattern;
799 FAT_WORD *patptr;
800 {
801     extern u_long *Host_memory;
802     register u_long i, *mem;
803
804     if (!Host) return pec_write_pattern(pattern, patptr);
805
806     /* Check if pattern number is valid */
807     if((pattern < 0) || (pattern > 0x40000))
808         return(FAILURE);
809
810     mem = 4(Host_memory[4 * pattern]);
811
812     /* Read pattern word */
813     for (i = 0; i < 3; ++i)
814         *mem++ = patptr->mem_bank[i];
815     return(SUCCESS);
816 }
817
818 pel_build_pattern_control(first_pattern_no, pattern_count, mode)
819 {
820     if (!Host) return SUCCESS;
821     return build_pattern_control(first_pattern_no, pattern_count, mode);
822 }
823
824
825 char *Soft_patterns[2] = {
826     "z2f5f554c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c54c5
```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pe1_pattern.c | DATE 5/23/89 | PAGE # 8/137 |
|--|--|---------------------------------------|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 841 | | | | |
| 842 | | | | |
| 843 | | | | |
| 844 | | | | |
| 845 | | | | |
| 846 | | | | |
| 847 | | | | |
| 848 | | | | |
| 849 | | | | |
| 850 | | | | |
| 851 | | | | |
| 852 | | | | |
| 853 | | | | |
| 854 | | | | |
| 855 | | | | |
| 856 | | | | |
| 857 | | | | |
| 858 | | | | |
| 859 | | | | |
| 860 | | | | |
| 861 | | | | |
| 862 | | | | |
| 863 | | | | |
| 864 | | | | |
| 865 | | | | |
| 866 | | | | |
| 867 | | | | |
| 868 | | | | |
| 869 | | | | |
| 870 | | | | |
| 871 | | | | |
| 872 | | | | |
| 873 | | | | |
| 874 | | | | |
| 875 | | | | |
| 876 | | | | |
| 877 | | | | |
| 878 | | | | |
| 879 | | | | |
| 880 | | | | |
| 881 | | | | |
| 882 | | | | |
| 883 | | | | |
| 884 | | | | |
| 885 | | | | |
| 886 | | | | |
| 887 | | | | |
| 888 | | | | |
| 889 | | | | |
| 890 | | | | |
| 891 | | | | |
| 892 | | | | |
| 893 | | | | |
| 894 | | | | |
| 895 | | | | |
| 896 | | | | |
| 897 | | | | |
| 898 | | | | |
| 899 | | | | |
| 900 | | | | |
| 901 | | | | |
| 902 | | | | |
| 903 | | | | |
| 904 | | | | |
| 905 | | | | |
| 906 | | | | |
| 907 | | | | |
| 908 | | | | |
| 909 | | | | |
| 910 | | | | |
| 911 | | | | |
| 912 | | | | |
| 913 | | | | |
| 914 | | | | |
| 915 | | | | |
| 916 | | | | |
| 917 | | | | |
| 918 | | | | |
| 919 | | | | |
| 920 | | | | |
| 921 | | | | |
| 922 | | | | |
| 923 | | | | |
| 924 | | | | |
| 925 | | | | |
| 926 | | | | |
| 927 | | | | |
| 928 | | | | |
| 929 | | | | |
| 930 | | | | |
| 931 | | | | |
| 932 | | | | |
| 933 | | | | |
| 934 | | | | |
| 935 | | | | |
| 936 | | | | |
| 937 | | | | |
| 938 | | | | |
| 939 | | | | |
| 940 | | | | |
| 941 | | | | |
| 942 | | | | |
| 943 | | | | |
| 944 | | | | |
| 945 | | | | |
| 946 | | | | |
| 947 | | | | |
| 948 | | | | |
| 949 | | | | |
| 950 | | | | |
| 951 | | | | |
| 952 | | | | |
| 953 | | | | |
| 954 | | | | |
| 955 | | | | |
| 956 | | | | |
| 957 | | | | |
| 958 | | | | |
| 959 | | | | |
| 960 | | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel_pattern.c

DATE 5/23/89
TIME 4:41:28 pm

PAGE #
9/138

```

LINE #          SOURCE TEXT
961      ln_message("VSE = 45.2F", vsh);
962      }
963      ln_message("  DUTVCC = 45.2F %0000\n", dutvcc);
964
965      for (bit = 8; bit < 12; bit++) {
966          if (find_voltage(bit, &voltage, &dutvcc) == FAILURE) return(FAILURE);
967          if (value == direction) {
968              current = (dutvcc - voltage) / Resistance(bit) * 1000.0;
969          } else {
970              current = voltage / Resistance(bit) * 1000.0;
971          }
972          ln_message("bit %2d, voltage = 45.2F, current = 45.2F, resistance = 44.1F\n",
973                  bit, voltage, current, Resistance(bit) / 1000.0);
974      }
975      return(SUCCESS);
976  }
977
978
979
980
981  pel_eval_soft_drivers()
982  {
983      register PEL *pel
984      = (PEL *) (pel_addr(current_lase, current_pel));
985      int magic, value, soft_drivers;
986
987      if (diag_clear_errors() != SUCCESS)
988          return(FAILURE);
989
990      if (pel_diag_dab_test_init(PUBLIC_DAB) != SUCCESS) {
991          (void) pel_err("Unable to initialize PEL/DAB for Soft-Driver evaluation.\n");
992          return(FAILURE);
993      }
994
995      pel->car.bit.eeprom_sel = 1;          /* takes on resistors on DIAGDAB */
996
997      for (magic = 0; magic < 5; magic++) {
998          ln_message("\n***** Testing MAGIC[%x] *****\n", magic);
999          for (value = 0; value < 2; value++) {
1000              for (soft_drivers = 1; soft_drivers < 16; soft_drivers++) {
1001                  ln_message("Testing % Soft-Drivers OutX\n", soft_drivers);
1002                  if (load_drive_soft(magic, value, soft_drivers) == FAILURE) return(FAILURE);
1003                  if (eval_current(value, FORWARD, 0.0, 0.0) == FAILURE) return(FAILURE);
1004                  if (eval_current(value, REVERSE, 0.4, 2.9) == FAILURE) return(FAILURE);
1005              }
1006          }
1007      }
1008      return(SUCCESS);
1009  }

```

Copyright 1989
Logic Modeling Systems

HEADER FILE
diags/pel_preamble.h

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:29 pm | 1/139 |

| LINE # | HEADER TEXT |
|--------|--|
| 1 | /* SCCS ID: pel_preamble.h Rev 3.1, 4/24/89 at 07:50:22 */ |
| 2 | |
| 3 | typedef struct { |
| 4 | /* during the first real pattern */ |
| 5 | unsigned :4, |
| 6 | :1, /* needed because MC will not turn on properly */ |
| 7 | hdns:1, |
| 8 | :2, /* needed to initialize SDOE/WDOE */ |
| 9 | seqhd:1, |
| 10 | seqsd:1, |
| 11 | lrychd:1, |
| 12 | lrycnd:1, |
| 13 | hdDis:1, |
| 14 | doff:1, |
| 15 | set:2, |
| 16 | reset:2, |
| 17 | ctl_out:1, |
| 18 | load_adns:1, /* used to load the SDEN/FORMAT registers */ |
| 19 | adns:4, |
| 20 | toggle:1, |
| 21 | format:2, |
| 22 | select_pel:1, |
| 23 | } PEL_PREAMBLE; |
| 24 | |
| 25 | typedef union { |
| 26 | PEL_PREAMBLE b; |
| 27 | unsigned long x; |
| 28 | } preamble; |
| 29 | |
| 30 | #define PREAMBLE_SIZE 28 |
| 31 | |

[illegible]

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel_sst.c

DATE 5/23/89
TIME 4:41:29 pm

PAGE #
2/141

```

121 if (pel_short_sensor_measure() != SUCCESS) {
122     pel_error("Short sensor measure failed\n");
123     retval = FAILURE;
124 }
125 return retval;
126 }
127
128 pel_short_byte(byte, trip)
129 {
130     int retval = SUCCESS, time_out, play_status;
131     register long chip, bit, pin;
132     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
133
134     /* long pulse should make sensor trip */
135     if ((time_out = pec_play_play()) == 0)
136     {
137         (void) pel_error("Unable to prepare for play.\n");
138         return(FAILURE);
139     }
140
141     for (chip = 0; chip < 5; ++chip) {
142         for (bit = 0; bit < 8; ++bit) {
143             pin = chip * 16 + byte * 8 + bit;
144
145             toggle_pattern_bit(22, pin); /* turn off SEQMD for shorted pin */
146             toggle_pattern_bit(28, pin);
147
148             /* clear pel errors */
149             pel_clear_pel_errors();
150
151             /* disable backplane errors */
152             pel->car.bit.magic_error_enable = 1;
153
154             play_status = pec_play(time_out);
155             if ((pel_sst_delay & 1))
156                 use_play_till_busy();
157             if (play_status == FAILURE) {
158                 (void) ls_error("Lane %c Pel %d: Play error trip %d byte %d chip %d bit %d\n",
159                     current_lane + 'A', current_pel,
160                     trip, byte, chip, bit);
161                 pec_play_cleanup();
162                 return(FAILURE);
163             }
164
165             if (report_by_error()) {
166                 (void) ls_error("Lane %c Pel %d: Unexpected backplane error from pin %d play\n",
167                     current_lane + 'A', current_pel,
168                     pin);
169                 return(FAILURE);
170             }
171             if (check_short_status(pin, trip) != SUCCESS) {
172                 retval = FAILURE;
173                 if (ls_error("Lane %c Pel %d Chip %d: Pin %d short sensor %s\n",
174                     current_lane + 'A', current_pel, chip,
175                     pin,
176                     trip? "did not trip" : "tripped") != SUCCESS)
177                     return FAILURE;
178             }
179             toggle_pattern_bit(22, pin); /* turn back on SEQMD after shorting */
180             toggle_pattern_bit(28, pin);
181         }
182     }
183     return retval;
184 }
185
186 set_pattern_bit(pattern, pin, value)
187 {
188     /*
189     Bank 0      Bank 1      Bank 2
190     +-----+ +-----+ +-----+
191     | 31 | 0 31 | 0 31 | 0 31 |
192     | 1  | 1  | 1  | 1  | 1  |
193     +-----+ +-----+ +-----+
194     | pin 79 | ..... | pin 0 | control |
195     +-----+ +-----+ +-----+
196     */
197     PAT_WORD patptr;
198     register long bank = 2 - ((pin + 16) >> 5);
199     register long mask = 1 << ((pin + 16) & 0x1f);
200
201     (void) pec_read_pattern(pattern, &patptr);
202
203     patptr.mem_bank[bank] = (value & 1)?
204     patptr.mem_bank[bank] & mask:
205     patptr.mem_bank[bank] & ~mask;
206
207     (void) pec_write_pattern(pattern, &patptr);
208 }
209
210 toggle_pattern_bit(pattern, pin)
211 {
212     /*
213     Bank 0      Bank 1      Bank 2
214     +-----+ +-----+ +-----+
215     | 31 | 0 31 | 0 31 | 0 31 |
216     | 1  | 1  | 1  | 1  | 1  |
217     +-----+ +-----+ +-----+
218     | pin 79 | ..... | pin 0 | control |
219     +-----+ +-----+ +-----+
220     */
221     PAT_WORD patptr;
222     register long bank = 2 - ((pin + 16) >> 5);
223     register long mask = 1 << ((pin + 16) & 0x1f);
224     register long value;
225
226     (void) pec_read_pattern(pattern, &patptr);
227
228     value = ((patptr.mem_bank[bank] & mask) == 0);
229
230     patptr.mem_bank[bank] = (value & 1)?
231     patptr.mem_bank[bank] & mask:
232     patptr.mem_bank[bank] & ~mask;
233
234     (void) pec_write_pattern(pattern, &patptr);
235 }
236
237 check_short_status(pin, trip)
238 {
239     register long retval;
240     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
241
242     if (pel->car.bit.magic_error == 0) {
243         retval = identify_shorted_pin(trip? pin: -1); /* short sample register */
244     }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pel_sst.c

DATE

5/23/89

PAGE #

TIME

4:41:29 pm

3/142

```

LINE #          SOURCE TEXT
241      } else retval = trip? FAILURE : SUCCESS;
242
243      /* clear pel errors */
244      pel_clear_pel_errors();
245
246      /* enable backplane errors */
247      pel->car.bit.magic_error_enable = 0;
248
249      return retval;
250
251  }
252
253  /* identify_shorted_pin returns SUCCESS iff the one pin tripped */
254  identify_shorted_pin(pin)
255  {
256      register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
257      register long chip, pinchip, bit;
258      short word, pinword, error;
259      long error_pin, retval = SUCCESS;
260
261      if (pel_sst_debug & 4) {
262          for (chip = 0; chip < 5; ++chip) {
263              lm_message("test ", pel->magic_chip[chip].reg[8] & 0xffff);
264              lm_message("\n");
265          }
266      }
267
268      if (pin != -1) {
269          pinchip = pin >> 4;
270          pinword = (1 << (pin & 0xf));
271      } else {
272          pinchip = 0;
273          pinword = 0;
274      }
275
276      for (chip = 0; chip < 5; ++chip) {
277          word = pel->magic_chip[chip].reg[8];
278          if (!(error = (chip == pinchip)? word ^ pinword : word))
279              continue;
280          retval = FAILURE;
281          for (bit = 0; bit < 16; ++bit) {
282              if (error & (1 << bit)) {
283                  error_pin = bit + 16 * chip;
284                  if (error_pin != pin) {
285                      if (lm_error("Lane %c Pel %d Pin %d (chip %d bit %d) shorted on pin %d test.\n",
286                          current_lane + 'A', current_pel,
287                          error_pin,
288                          chip, bit,
289                          pin) == FAILURE)
290                          return FAILURE;
291                  } else {
292                      if (lm_error("Lane %c Pel %d Pin %d (chip %d bit %d) did not trip.\n",
293                          current_lane + 'A', current_pel,
294                          error_pin, chip, bit) == FAILURE)
295                          return FAILURE;
296                  }
297              }
298          }
299      }
300      return retval;
301  }
302
303  pel_reset_all_pels_in_lane()
304  {
305      register PEL *pel;
306      register int pelno;
307
308      for (pelno=0; pelno < NUMBER_OF_PELS; ++pelno) {
309          if (probe_pel(current_lane, pelno) == SUCCESS) {
310              pel = (PEL *) (pel_addr(current_lane, pelno));
311              pel->car.bit.resetL = 0;
312          }
313      }
314      lm_message("\n");
315  }
316
317  pel_clear_pel_errors()
318  {
319      short dummy;
320      register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
321      /* clear pel errors */
322
323      dummy = pel->magic_chip[0].reg[15];
324      pel->car.bit.resetL = 0;
325      pel->car.bit.resetL = 1;
326      return (int) dummy; /* short trip time */
327  }
328
329  /* the following functions are used to measure short trip time */
330  pel_short_sensor_measure()
331  {
332      long byte, timeout, retval = SUCCESS;
333      register char *pattern_string;
334      long chip, bit, pin, time;
335      long min_time, max_time;
336
337      if (pel_lane_init() != SUCCESS) { /* set up pec/tmg/pel/dab */
338          (void) lm_error("Unable to initialize lane %c\n", current_lane + 'A');
339          return(FAILURE);
340      }
341
342      if (pel_dab_init(PUBLIC_DAB) != SUCCESS) {
343          (void) lm_error("Unable to initialize DAB at Lane %c, PEL %d\n",
344              current_lane + 'A', current_pel);
345          return(FAILURE);
346      }
347
348      if (set_vlog1(0.8) == FAILURE) return(FAILURE);
349      if (set_vlog2(2.0) == FAILURE) return(FAILURE);
350      if (set_vlth(0.3) == FAILURE) return(FAILURE);
351      if (set_val(0.4) == FAILURE) return(FAILURE);
352      if (set_vsb(4.0) == FAILURE) return(FAILURE);
353      if (set_vhth(4.4) == FAILURE) return(FAILURE);
354
355      min_time = 10000000; /* impossibly large */
356      max_time = 0; /* impossibly small */
357      lm_message("Measuring trip times");
358      for (byte = 0; byte < 2; ++byte) {
359          pattern_string = byte?
360

```


Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pel_util.c

DATE

5/23/89

PAGE #

TIME

4:41:30 pm

1/144

```

LINE # SOURCE TEXT
1  /* SCOS_ID: pel_util.c rev 3.1, 4/24/89 at 07:50:38 */
2  /*
3  * .....
4  *
5  * PEL utilities
6  * used in PEL diag routines
7  *
8  * .....
9  */
10 #include <math.h>
11 #include "common.h"
12 #include "mod_def.h"
13 #include "modeler_exta.h"
14 #include "exta.h"
15 #include "lm_diags.h"
16 #include "tmg.h"
17 #include "tmg_exta.h"
18 #include "tmg_def.h"
19 #include "pac.h"
20 #include "pac_def.h"
21 #include "pac_exta.h"
22 #include "magic.h"
23 #include "pel.h"
24
25 /*
26 * void flush_key_buf()
27 *
28 * INPUT: none
29 * OUTPUT: none
30 * DESCRIPTION: Clears key buffer.
31 */
32 flush_key_buf()
33 {
34     while(lm_check_key() != 0)
35         lm_get_key();
36 }
37
38 /*
39 * pel_tmg_init(void)
40 *
41 * This function initializes the Timing Generator to default
42 * settings.
43 *
44 * Inputs: none
45 * Outputs: function returns SUCCESS or FAILURE
46 */
47 pel_tmg_init()
48 {
49     long edgetime[6];
50
51     lm_message("\nInitializing Timing Generator to Defaults\n");
52     if (tmg_restore_calibration() != SUCCESS) {
53         return FAILURE;
54     }
55     /*
56     * These timings are absolutely critical to a number of different tests. They
57     * should never get changed!!!
58     */
59     edgetime[0] = 5000;
60     edgetime[1] = 5000;
61     edgetime[2] = 5000;
62     edgetime[3] = 15000;
63     edgetime[4] = 5000;
64     edgetime[5] = 0;
65     if (tmg_set_timings(40000L, edgetime, 0L, 100000L, 8000000L) != SUCCESS) {
66         return FAILURE;
67     }
68     lane_select(lane_code(current_lane));
69     lm_delay(1); /* wait for PEL to lock */
70     return(SUCCESS);
71 }
72
73 /*
74 * pel_pac_init(void)
75 *
76 * This function initializes the PAC
77 *
78 * Inputs: none
79 * Outputs: function returns SUCCESS or FAILURE
80 */
81 pel_pac_init()
82 {
83     if (pac[current_lane].exists == TRUE) {
84         if (pac_stack_pace(current_lane) != SUCCESS) {
85             (void)pel_error("Unable to set up pattern memory.\n");
86             return(FAILURE);
87         }
88     } else {
89         (void)pel_error("There is no Pattern Controller in this lane.\n");
90         return(FAILURE);
91     }
92     Clear_pac_errors(current_lane);
93     return(SUCCESS);
94 }
95
96 /*
97 * pel_dab_init()
98 *
99 * This function initializes the DAB
100 *
101 * Inputs: none
102 * Outputs: function returns SUCCESS or FAILURE
103 */
104 pel_dab_init(mode)
105 int mode;
106 {
107     register PEL *pel;
108     = (PEL *) (pel_addr(current_lane, current_pel));
109     u_short dummy;
110     dummy = pel->magic_chip[0].reg[15]; /* reset MAGIC chip errors */
111     pel->car.bit.resetl = 0; /* reset the PEL */
112     pel->car.bit.initialize = 0;
113     pel->car.bit.eeprom_in = 0;
114     pel->car.bit.eeprom_clk = 0;
115     pel->car.bit.eeprom_sel = 0;
116 }

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/pel_util.c | DATE 5/23/89 | PAGE # 2/145 |
|--|--|------------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 121 | pel->car.bit.magic_error_enable = 0; | | | |
| 122 | pel->car.bit.is_woe_led = 1; | | | |
| 123 | pel->car.bit.initialize = 1; | | | |
| 124 | pel->car.bit.reset1 = 1; | | | |
| 125 | if (mode == PUBLIC_DAB) { | | | |
| 126 | pel->car.bit.private = 0; | | | |
| 127 | if (pel->car.reg != 0x1f13) { | | | |
| 128 | (void)pel_error("Unable to initialize PUBLIC DAB\n"); | | | |
| 129 | return(FAILURE); | | | |
| 130 | } | | | |
| 131 | } else { /* mode = PRIVATE */ | | | |
| 132 | pel->car.bit.private = 1; | | | |
| 133 | if (pel->car.reg != 0x1f17) { | | | |
| 134 | (void)pel_error("Unable to initialize PRIVATE DAB\n"); | | | |
| 135 | return(FAILURE); | | | |
| 136 | } | | | |
| 137 | } | | | |
| 138 | if (dummy), /* shut lint up */ | | | |
| 139 | return(SUCCESS); | | | |
| 140 | } | | | |
| 141 | /* pel_lase_init(void) | | | |
| 142 | /* This function initializes the lase (and the current PEL) | | | |
| 143 | /* Inputs: none | | | |
| 144 | /* Outputs: function returns SUCCESS or FAILURE | | | |
| 145 | */ | | | |
| 146 | pel_lase_init() | | | |
| 147 | { | | | |
| 148 | /* Clear any PEL errors which may be present */ | | | |
| 149 | pel_clear_pel_errors(); | | | |
| 150 | if (pel_tmg_init() != SUCCESS) { | | | |
| 151 | (void)pel_error("Unable to initialize TMS\n"); | | | |
| 152 | return(FAILURE); | | | |
| 153 | } | | | |
| 154 | if (pel_pcc_init() != SUCCESS) { | | | |
| 155 | (void)pel_error("Unable to initialize PAC\n"); | | | |
| 156 | return(FAILURE); | | | |
| 157 | } | | | |
| 158 | return(SUCCESS); | | | |
| 159 | } | | | |
| 160 | more(st, size, count) | | | |
| 161 | { | | | |
| 162 | register long illegal; | | | |
| 163 | static char sol[] = "a b c d e f g h i j k l m n o p q r s t u v w x y z"; /* start of line */ | | | |
| 164 | /* message = "More--(addr)", 100*(1+st)/(1+size); | | | |
| 165 | do { | | | |
| 166 | illegal = 0; | | | |
| 167 | switch (lm_get_key()) { | | | |
| 168 | case 'q': lm_message("table\n", sol, ""); return 0; break; | | | |
| 169 | case 'a': --count; lm_message("ts", sol); break; | | | |
| 170 | case 'i': lm_message("is", sol); break; | | | |
| 171 | default: lm_message("007", &illegal, /* illegal char */ | | | |
| 172 | Clear key buf(); | | | |
| 173 | while (illegal); | | | |
| 174 | return count; | | | |
| 175 | } | | | |
| 176 | } while (illegal); | | | |
| 177 | return count; | | | |
| 178 | } | | | |
| 179 | #define is_space(c) (((c) == ' ') ((c) == '\t')) | | | |
| 180 | #define is_num(c) (((c) >= '0') && ((c) <= '9')) | | | |
| 181 | char * | | | |
| 182 | get_range(s, from, to) | | | |
| 183 | register char *s; | | | |
| 184 | register u_long *from, *to; | | | |
| 185 | { | | | |
| 186 | char buffer[128]; | | | |
| 187 | register char *b; | | | |
| 188 | while (is_space(*s)) ++s; | | | |
| 189 | b = buffer; | | | |
| 190 | if (!is_num(*s)) return 0; /* syntax error */ | | | |
| 191 | while (is_num(*s)) *b++ = *s++; | | | |
| 192 | *b = '\0'; | | | |
| 193 | *from = *to = atoi(buffer); | | | |
| 194 | while (is_space(*s)) ++s; | | | |
| 195 | if (*s == '-') { | | | |
| 196 | /* Range of values */ | | | |
| 197 | ++s; | | | |
| 198 | while (is_space(*s)) ++s; | | | |
| 199 | b = buffer; | | | |
| 200 | if (!is_num(*s)) return 0; /* syntax error */ | | | |
| 201 | while (is_num(*s)) *b++ = *s++; | | | |
| 202 | *b = '\0'; | | | |
| 203 | *to = atoi(buffer); | | | |
| 204 | while (is_space(*s)) ++s; | | | |
| 205 | } | | | |
| 206 | if (*s == ',') ++s; /* optional comma separator */ | | | |
| 207 | return s; | | | |
| 208 | } | | | |
| 209 | pel_set_timing() | | | |
| 210 | { | | | |
| 211 | static long edgetime[6] = { | | | |
| 212 | 10000, 10000, 10000, 15000, 10000, 0 | | | |
| 213 | }; | | | |
| 214 | static long period = 40000; | | | |
| 215 | long edge; | | | |
| 216 | char buffer[80]; | | | |
| 217 | diag_get_long(&period, "Period", 40000, 0x7fffffff); | | | |
| 218 | for (edge = 0; edge < 6; ++edge) { | | | |
| 219 | sprintf(buffer, "Edge %d", edge); | | | |
| 220 | } | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/pel_util.c

DATE 5/23/89
TIME 4:41:30 pm

PAGE #
3/146

```

LINE #          SOURCE TEXT
241      diag_get_logg(&edgetime[edge]), buffer, 01, period);
242      }
243      if (tag_set_timing(period, edgetime, 01, 1000001, 25000001) != SUCCESS) {
244          return FAILURE;
245      }
246      return SUCCESS;
247  }
248  }
249  pel_set_voltages()
250  {
251      long vlogl = 800;
252      long vlogh = 2000;
253      long vlth = 300;
254      long val = 400;
255      long vah = 4000;
256      long vlth = 4400;
257
258      do {
259          diag_get_logg(&vlogl, "vlogl (mv)", 01, 120001);
260          } while (set_vlogl(0.001 * (float)(vlogl)) != SUCCESS);
261      do {
262          diag_get_logg(&vlogh, "vlogh (mv)", 01, 120001);
263          } while (set_vlogh(0.001 * (float)(vlogh)) != SUCCESS);
264      do {
265          diag_get_logg(&vlth, "vlth (mv)", 01, 120001);
266          } while (set_vlth(0.001 * (float)(vlth)) != SUCCESS);
267      do {
268          diag_get_logg(&val, "val (mv)", 01, 120001);
269          } while (set_val(0.001 * (float)(val)) != SUCCESS);
270      do {
271          diag_get_logg(&vah, "vah (mv)", 01, 120001);
272          } while (set_vah(0.001 * (float)(vah)) != SUCCESS);
273      do {
274          diag_get_logg(&vlth, "vlth (mv)", 01, 120001);
275          } while (set_vlth(0.001 * (float)(vlth)) != SUCCESS);
276      }
277  }
278  pel_loop_play()
279  {
280      int time_out;
281      flush_key_buf();
282
283      if ((time_out = pac_pre_play()) == 0)
284      {
285          (void)pel_error("Unable to prepare for looping play.\n");
286          return FAILURE;
287      }
288      in_message("Hit key to stop...");
289
290      while (!in_check_key()) {
291          if (pac_play(time_out) == FAILURE) {
292              (void)pel_error("pel_play_pattern_string failed in pac_play.\n");
293              pac_play_cleanup();
294              break;
295          }
296      }
297      flush_key_buf();
298      return SUCCESS;
299  }
300
301

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/probe.c | DATE 5/23/89 TIME 4:41:30 pm | PAGE # 1/147 |
|--|---|---------------------------------|---------------------------------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCSS ID: probe.c rev 3.1, 4/24/89 at 07:50:31 */ | | | |
| 2 | /*.....*/ | | | |
| 3 | /* probe.c */ | | | |
| 4 | /* | | | |
| 5 | /* Probe Function | | | |
| 6 | /* used in diagnostics */ | | | |
| 7 | /*.....*/ | | | |
| 8 | /*.....*/ | | | |
| 9 | /*.....*/ | | | |
| 10 | /*.....*/ | | | |
| 11 | #include <math.h> | | | |
| 12 | #include "common.h" | | | |
| 13 | #include "lm_diags.h" | | | |
| 14 | #include "mod_def.h" | | | |
| 15 | #include "vrtx.h" | | | |
| 16 | #include "tmg.h" | | | |
| 17 | #include "pac.h" | | | |
| 18 | #include "magic.h" | | | |
| 19 | #include "pel.h" | | | |
| 20 | | | | |
| 21 | int | | | |
| 22 | probe_tmg() | | | |
| 23 | { | | | |
| 24 | if(lm_read_probe((long)(0x80000000)) != SUCCESS) | | | |
| 25 | { | | | |
| 26 | (void)lm_warning("No Timing Generator installed.\n"); | | | |
| 27 | return(FAILURE); | | | |
| 28 | } | | | |
| 29 | else | | | |
| 30 | return(SUCCESS); | | | |
| 31 | } | | | |
| 32 | | | | |
| 33 | int | | | |
| 34 | probe_all_lanes() | | | |
| 35 | { | | | |
| 36 | int lane_no; | | | |
| 37 | | | | |
| 38 | /* Reset backplane lanes */ | | | |
| 39 | if(diag_tmg_reset(TRUE) != SUCCESS) | | | |
| 40 | { | | | |
| 41 | (void)lm_error("Could not reset backplane during lane probe.\n"); | | | |
| 42 | return(FAILURE); | | | |
| 43 | } | | | |
| 44 | for(lane_no = 0; lane_no < NUMBER_OF_LANES; lane_no++) | | | |
| 45 | { | | | |
| 46 | if(probe_lane(lane_no) == SUCCESS) | | | |
| 47 | return(SUCCESS); | | | |
| 48 | } | | | |
| 49 | (void)lm_warning("There are no PACs or PELs installed.\n"); | | | |
| 50 | return(FAILURE); | | | |
| 51 | } | | | |
| 52 | | | | |
| 53 | int | | | |
| 54 | probe_lane(lane_no) | | | |
| 55 | { | | | |
| 56 | int lane_no; | | | |
| 57 | | | | |
| 58 | if((probe_pac(lane_no) == SUCCESS) (probe_all_pels(lane_no) == SUCCESS)) | | | |
| 59 | return(SUCCESS); | | | |
| 60 | else | | | |
| 61 | return(FAILURE); | | | |
| 62 | } | | | |
| 63 | | | | |
| 64 | int | | | |
| 65 | probe_pac(lane_no) | | | |
| 66 | { | | | |
| 67 | int lane_no; | | | |
| 68 | | | | |
| 69 | if(lm_read_probe((long)(LANE_A_OFFSET + (lane_no * LANE_SIZE) + | | | |
| 70 | PAC_REC_OFFSET)) != SUCCESS) | | | |
| 71 | return(FAILURE); | | | |
| 72 | else | | | |
| 73 | return(SUCCESS); | | | |
| 74 | } | | | |
| 75 | | | | |
| 76 | int | | | |
| 77 | probe_all_pels(lane_no) | | | |
| 78 | { | | | |
| 79 | int lane_no; | | | |
| 80 | | | | |
| 81 | int slot_no; | | | |
| 82 | | | | |
| 83 | for(slot_no = 0; slot_no < NUMBER_OF_SLOTS; slot_no++) | | | |
| 84 | { | | | |
| 85 | if(probe_pel(lane_no, slot_no) == SUCCESS) | | | |
| 86 | return(SUCCESS); | | | |
| 87 | } | | | |
| 88 | return(FAILURE); | | | |
| 89 | } | | | |
| 90 | | | | |
| 91 | int | | | |
| 92 | probe_pel(lane_no, slot_no) | | | |
| 93 | { | | | |
| 94 | int lane_no; | | | |
| 95 | int slot_no; | | | |
| 96 | | | | |
| 97 | if(lm_read_probe((long)(pel_addr(lane_no, slot_no))) != SUCCESS) | | | |
| 98 | return(FAILURE); | | | |
| 99 | else | | | |
| 100 | return(SUCCESS); | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_cal.c | DATE 5/23/89 | PAGE # 1/148 |
|--|---|-----------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCOS_ID: tmg_cal.c rev 3.2, 5/9/89 at 15:54:16 */ | | | |
| 2 | /*-----*/ | | | |
| 3 | /* | | | |
| 4 | /* Calib routines for the Timing Generator. */ | | | |
| 5 | /*-----*/ | | | |
| 6 | /* | | | |
| 7 | /*-----*/ | | | |
| 8 | /* | | | |
| 9 | #include <math.h> | | | |
| 10 | #include "common.h" | | | |
| 11 | #include "mod_def.h" | | | |
| 12 | #include "modeler_extn.h" | | | |
| 13 | #include "vrtx.h" | | | |
| 14 | #include "tmg.h" | | | |
| 15 | #include "tmg_def.h" | | | |
| 16 | #include "tmg_extn.h" | | | |
| 17 | #include "tmg_run.h" | | | |
| 18 | #include "pac.h" | | | |
| 19 | #include "pac_def.h" | | | |
| 20 | #include "pac_extn.h" | | | |
| 21 | #ifdef DIAGS | | | |
| 22 | #include "diag_setjmp.h" | | | |
| 23 | #endif DIAGS | | | |
| 24 | #include "intr.h" | | | |
| 25 | /* | | | |
| 26 | #ifdef DIAGS | | | |
| 27 | #define lm_message printf | | | |
| 28 | #define lm_warning printf | | | |
| 29 | #define lm_error printf | | | |
| 30 | null function({}) | | | |
| 31 | #endif DIAGS | | | |
| 32 | /* | | | |
| 33 | #ifdef HOST | | | |
| 34 | #include "lm_diags.h" | | | |
| 35 | #define lm_message lm_message | | | |
| 36 | #define lm_error (++total_errors[current_depth], lm_message) | | | |
| 37 | #define lm_warning lm_message | | | |
| 38 | static long current_depth = 0; | | | |
| 39 | #endif HOST | | | |
| 40 | /* | | | |
| 41 | #undef MIN_PERIOD | | | |
| 42 | #define MIN_PERIOD 33333 | | | |
| 43 | #define BIN_LIMIT 100 /* binary search count limit */ | | | |
| 44 | /* | | | |
| 45 | static void tmg_assume_mint() | | | |
| 46 | static long tmg_lowest_delay_edge(); | | | |
| 47 | /* | | | |
| 48 | long tmg_cal_debug = 0; | | | |
| 49 | long tmg_reduction_factor = 82; /* % reduction for edge search */ | | | |
| 50 | /* | | | |
| 51 | #define tmg_detect_always_high(x, e, p, t, c, actual) \ | | | |
| 52 | tmg_detect_edge(x, e, DETECT_BOOL, \ | | | |
| 53 | p, t, c, actual) | | | |
| 54 | /* | | | |
| 55 | #define tmg_detect_always_low(x, e, p, t, c, actual) \ | | | |
| 56 | tmg_detect_edge(x, e, DETECT_BOOL DETECT_LOW, \ | | | |
| 57 | p, t, c, actual) | | | |
| 58 | /* | | | |
| 59 | #define tmg_detect_sample_always_high(xr, sr, p, t7, ts, c, actual) \ | | | |
| 60 | tmg_detect_sample(xr, sr, DETECT_BOOL DETECT_SAMPLE, \ | | | |
| 61 | p, t7, ts, c, actual) | | | |
| 62 | /* | | | |
| 63 | #define tmg_detect_sample_always_low(xr, sr, p, t7, ts, c, actual) \ | | | |
| 64 | tmg_detect_sample(xr, sr, \ | | | |
| 65 | DETECT_BOOL DETECT_LOW DETECT_SAMPLE, \ | | | |
| 66 | p, t7, ts, c, actual) | | | |
| 67 | /* | | | |
| 68 | #define tmg_g_t_ramp() (tmgptr->sample_delay_range) | | | |
| 69 | #define tmg_set_ramp(ramp) (tmgptr->sample_delay_range = (ramp)) | | | |
| 70 | /* | | | |
| 71 | #define CALIBRATION_DONE 0xDEADBEF | | | |
| 72 | /* | | | |
| 73 | /* XXXX */ | | | |
| 74 | #undef tmg_set_threshold | | | |
| 75 | /* | | | |
| 76 | tmg_set_threshold(edge, threshold) | | | |
| 77 | int edge, threshold; | | | |
| 78 | { | | | |
| 79 | int i, other_thresh; | | | |
| 80 | if(threshold < 128) | | | |
| 81 | other_thresh = 255; | | | |
| 82 | else | | | |
| 83 | other_thresh = 10; | | | |
| 84 | tmgptr->edge_delay[edge].delay = threshold; | | | |
| 85 | for(i = 0; i < 6; i++) | | | |
| 86 | { | | | |
| 87 | if(i == edge) | | | |
| 88 | continue; | | | |
| 89 | tmgptr->edge_delay[i].delay = other_thresh; | | | |
| 90 | } | | | |
| 91 | } | | | |
| 92 | /* | | | |
| 93 | tmg_calibrate() | | | |
| 94 | { | | | |
| 95 | return tmg_do_calibrate(1); | | | |
| 96 | } | | | |
| 97 | /* | | | |
| 98 | tmg_recalibrate() | | | |
| 99 | { | | | |
| 100 | return tmg_do_calibrate(0); | | | |
| 101 | } | | | |
| 102 | /* | | | |
| 103 | tmg_do_calibrate(destructive) | | | |
| 104 | { | | | |
| 105 | /* | | | |
| 106 | #ifdef info | | | |
| 107 | 1. reset the clock board | | | |
| 108 | 2. find the minimum threshold of the fast ramp for all edges | | | |
| 109 | 3. set min_thresh of ramp to ramp 0's value for all edges; | | | |
| 110 | 4. calculate slope & offset of fast ramp (0) for each edge | | | |
| 111 | 5. set the delay line | | | |
| 112 | 6. Finally calibrate all edges of all ramps | | | |
| 113 | 7. Next measure edge 7 and Sample's mint | | | |
| 114 | 8. Measure edge7 and sample ramps | | | |
| 115 | 9. Measure sample offsets | | | |
| 116 | 10. Measure Early mode sample offsets | | | |
| 117 | #endif info | | | |
| 118 | INTR_INIT | | | |
| 119 | extern struct CALIB default_calib; | | | |
| 120 | static char me[] = "tmg_calibrate"; | | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_cal.c

DATE

5/23/89

PAGE #

2/149

TIME

4:41:30 pm

```

121  register long ramp, aramp, eramp, first_aramp, last_aramp, edge,
122  long threshold, thresh7, thresha;
123  char buffer[80];
124
125  long errors = 0;
126
127  INTR_BEGIN
128
129  /* if destructive, we should do a backplane reset (diag_tmg_reset(TRUE); */
130  /* because a side effect of that is to do a par_info_init */
131  if(diag_tmg_reset(destructive ? TRUE : FALSE) != SUCCESS)
132  {
133      tmg_report_failure(me, "diag_tmg_reset");
134      ++errors;
135      goto cleanup;
136  }
137
138  /* find the minimum threshold of the fast ramp for all edges */
139  tmg_set_delay(14); /* delay = 6ns + 14ns */
140
141  tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE;
142
143  if (destructive) {
144      tmgptr->backplane_reset = 0; /* assert backplane reset */
145      lm_message("Measuring edge minimum thresholds");
146      for (edge = 0; edge < NUMBER_OF_EDGES; ++edge) {
147          lm_message("-");
148          if (tmg_measure_minth(01, edge, athreshold) != SUCCESS) {
149              tmg_report_failure(me, "tmg_measure_minth");
150              ++errors;
151              if (!(tmg_cal_debug & 0x20000000))
152                  goto cleanup;
153          }
154          calib.EdgeMinThresh[0][edge] = threshold + START_DEAD_TIME_FUDGE;
155      }
156      lm_message("Done\n");
157  } else {
158      if (!tmg_is_calibrated()) {
159          for (edge = 0; edge < NUMBER_OF_EDGES; ++edge) {
160              calib.EdgeMinThresh[0][edge] = default_calib.EdgeMinThresh[0][edge];
161          }
162      }
163  }
164
165  /* set min thresh of ramp to ramp 0's value for all edges */
166  tmg_set_minth(0calib);
167
168  lm_message("Measuring edge ramp slopes");
169
170  for (ramp = 0; ramp < NUMBER_OF_RAMPS; ++ramp) {
171      for (edge = 0; edge < NUMBER_OF_EDGES; ++edge) {
172          lm_message("-");
173          if (tmg_measure_ramp(ramp, edge, 0calib, &errors) != SUCCESS) {
174              sprintf(buffer, "tmg_measure_ramp(%d,%d)", ramp, edge);
175              tmg_report_failure(me, buffer);
176              ++errors;
177              if (!(tmg_cal_debug & 0x20000000))
178                  goto cleanup;
179          }
180      }
181  }
182
183  lm_message("Done\n");
184  #ifdef DIAGS
185      if (tmg_cal_debug & 0x40000000)
186          tmg_play_til_key();
187  #endif DIAGS
188
189  /* set the delay line */
190
191  if (tmg_calibrate_delay(0calib) != SUCCESS) {
192      tmg_report_failure(me, "tmg_calibrate_delay");
193      ++errors;
194      if (!(tmg_cal_debug & 0x20000000))
195          goto cleanup;
196  }
197  #ifdef DIAGS
198      if (tmg_cal_debug & 0x40000000)
199          tmg_play_til_key();
200  #endif DIAGS
201
202  /* remeasure the offsets */
203
204  lm_message("Measuring edge offsets");
205  for (ramp = 0; ramp < NUMBER_OF_RAMPS; ++ramp) {
206      for (edge = 0; edge < NUMBER_OF_EDGES; ++edge) {
207          if (tmg_measure_offset(ramp, edge, 0calib) != SUCCESS) {
208              tmg_report_failure(me, "tmg_measure_offset");
209              ++errors;
210              if (!(tmg_cal_debug & 0x20000000))
211                  goto cleanup;
212          }
213      }
214  }
215
216  lm_message("Done\n");
217  #ifdef DIAGS
218      if (tmg_cal_debug & 0x40000000)
219          tmg_play_til_key();
220  #endif DIAGS
221
222  if (destructive) {
223      /* Next measure edge 7 and sample's minth */
224      /* (edge 7 is the edge that triggers a SAMPLE) */
225      lm_message("Measuring Edge7 and Sample minimum thresholds");
226      ramp = 0;
227      tmg_set_sample_threshold(50); /* sure to cause a trigger */
228      tmg_set_edge7_threshold(50); /* sure to cause a trigger */
229      if (tmg_measure_edge7_sample_minth(01, 01, athresh7, lthresha) != SUCCESS) {
230          tmg_report_failure(me, "tmg_measure_edge7_sample_minth");
231          ++errors;
232          if (!(tmg_cal_debug & 0x20000000))
233              goto cleanup;
234      }
235      lm_message("Done\n");
236  }
237  #ifdef DIAGS
238      if (tmg_cal_debug & 0x40000000)
239          tmg_play_til_key();
240  #endif DIAGS
241
242  for(eramp = 0; eramp < NUMBER_OF_ERAMPS; eramp++)
243      calib.Edge7MinThresh[eramp] = thresh7 - START_DEAD_TIME_FUDGE;

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_cal.c

DATE 5/23/89

PAGE #

TIME 4:41:30 pm

3/150

```

LINE # SOURCE TEXT
241 for(aramp = 0; aramp < NUMBER_OF_SRAMPs; aramp++)
242     calib.SampleMinThresh(aramp) = threshs + START_DEAD_TIME_FUDGE;
243 } else {
244     if (!tmg_is_calibrated()) {
245         for(aramp = 0; aramp < NUMBER_OF_SRAMPs; aramp++)
246             calib.Edge7MinThresh(aramp) = Default_calib.Edge7MinThresh(aramp);
247         for(aramp = 0; aramp < NUMBER_OF_SRAMPs; aramp++)
248             calib.SampleMinThresh(aramp) = Default_calib.SampleMinThresh(aramp);
249     }
250 }
251 lm_message("Measuring Sample ramp slopes");
252 for (aramp = 0; aramp < NUMBER_OF_SRAMPs; ++aramp) {
253     lm_message(".");
254     if (tmg_measure_edge7_ramp(aramp, &calib) != SUCCESS) {
255         tmg_report_failure(me, "tmg_measure_edge7_ramp");
256         ++errors;
257         if (!(tmg_cal_debug & 0x20000000))
258             goto cleanup;
259     }
260     else {
261         switch (aramp) {
262             case 0:
263                 first_aramp = 0; /* just do first one */
264                 last_aramp = 0;
265                 break;
266             case 1:
267                 first_aramp = 1; /* do next 2 */
268                 last_aramp = 2;
269                 break;
270             case 2:
271                 first_aramp = 3; /* do last one */
272                 last_aramp = 3;
273                 break;
274             default:
275                 first_aramp = 100; /* don't do any */
276                 last_aramp = 0;
277                 break;
278         }
279         for(aramp = first_aramp; aramp <= last_aramp; aramp++) {
280             lm_message(".");
281             if (tmg_measure_sample_ramp(aramp, aramp, &calib, &errors)
282                 != SUCCESS) {
283                 tmg_report_failure(me, "tmg_measure_sample_ramp");
284                 ++errors;
285                 if (!(tmg_cal_debug & 0x20000000))
286                     goto cleanup;
287             }
288         }
289     }
290 }
291 lm_message("Done\n");
292 #ifdef DIAGS
293     if (tmg_cal_debug & 0x00000000)
294         tmg_play_til_key();
295 #endif DIAGS
296 lm_message("Measuring sample offsets");
297 for(aramp = 0; aramp < NUMBER_OF_SRAMPs; aramp++) {
298     for(aramp = 0; aramp < NUMBER_OF_SRAMPs; aramp++) {
299         lm_message(".");
300         if (tmg_measure_sample_offset_only(aramp, aramp, &calib,
301             EDGE7SAMPLETRIGGERMODE) != SUCCESS) {
302             tmg_report_failure(me, "tmg_measure_sample_offset_only\
303                 , Edge 7 mode");
304             ++errors;
305             if (!(tmg_cal_debug & 0x20000000))
306                 goto cleanup;
307         }
308     }
309 }
310
311
312
313 for(aramp = 0; aramp < NUMBER_OF_SRAMPs; aramp++)
314 {
315     if (tmg_measure_sample_offset_only(01.aramp, &calib,
316         EARLYSAMPLETRIGGERMODE) != SUCCESS) {
317         tmg_report_failure(me, "tmg_measure_sample_offset_only, early mode");
318         ++errors;
319         if (!(tmg_cal_debug & 0x20000000))
320             goto cleanup;
321     }
322     lm_message(".");
323 }
324
325 lm_message("Done\n");
326 #ifdef DIAGS
327     if (tmg_cal_debug & 0x00000000)
328         tmg_play_til_key();
329 #endif DIAGS
330
331 if (tmg_complete_calib_structure(&calib) != SUCCESS) {
332     tmg_report_failure(me, "tmg_complete_calib_structure");
333     ++errors;
334     if (!(tmg_cal_debug & 0x20000000))
335         goto cleanup;
336 }
337
338 if (tmg_cal_debug & 0x80000000)
339 {
340     if (tmg_print_calib_structure(&calib, != SUCCESS)
341         {
342             tmg_report_failure(me, "tmg_print_calib_structure");
343             ++errors;
344             if (!(tmg_cal_debug & 0x20000000))
345                 goto cleanup;
346         }
347     }
348
349 if (calib.EarlySampleMinDelay[0] > (long)calib.EdgeMinDelay[0])
350 {
351     ++errors;
352     lm_error("EarlySampleMinDelay[0] = %d ps > EdgeMinDelay[0] = %d ps\n",
353         calib.EarlySampleMinDelay[0], calib.EdgeMinDelay[0]);
354 }
355
356 if (tmg_cal_debug & 0x80000000)
357 {
358     if (tmg_print_data_logging_messages(&calib) != SUCCESS)
359         {
360

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_cal.cDATE 5/23/89
TIME 4:41:30 pmPAGE #
4/151

```

LINE # SOURCE TEXT
361 tmg_report_failure(mo, "tmg_print_data_logging_messages");
362 errors++;
363 if(!((tmg_cal_debug & 0x20000000)))
364     goto cleanup;
365 }
366
367 cleanup:
368
369 tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE;
370
371 tmg_set_test_mode(0);
372 if ((errors == 0) || ((tmg_cal_debug & 0x20000000) != 0)) {
373     tmg_save_calibration();
374     calib.CalCompleted = CALIBRATION_DONE;
375 }
376 if (destructive) {
377     (void)diag_tmg_reset(1); /* full reset */
378 }
379
380 INTR_GOT_ONE
381 tmg_set_test_mode(0);
382 calib.CalCompleted = 0; /* unsuccessful */
383 (void)diag_tmg_reset(0);
384 intr();
385 INTR_END
386
387 if (errors) {
388     calib.CalCompleted = 0; /* unsuccessful */
389     return(FAILURE);
390 }
391 return(SUCCESS);
392
393 /*
394 tmg_incremental_calibrate(part, destructiveFlag)
395 *
396 * Does a portion of what tmg_do_calibrate() does, selected by
397 * 'part'. Only does minimum threshold measurements if the
398 * 'destructiveFlag' is TRUE (normally only done at power up and
399 * at very long intervals, if ever again).
400 */
401 int tmg_incremental_calibrate(part, destructive)
402 int part, destructive;
403 {
404     static struct CALIB tmpcal;
405     int returncode = SUCCESS;
406     long threshold, thresh7, thresholds, edge, aramp, errors;
407     int save_delay_line;
408
409     if((part >= 0) && (part < 5) && !destructive)
410     {
411         edge = part & 4;
412         tmgptr->backplane_reset = 0; /* aspart backplane reset */
413         save_delay_line = tmg_get_delay();
414         tmg_set_delay(14);
415         if(tmg_measure_minth(01, edge, &threshold) != SUCCESS)
416             returncode = FAILURE;
417         else
418             for(aramp = 0; aramp < NUMBER_OF_ERAMPS; aramp++)
419                 calib.EdgeMinThreshold[aramp][edge] = threshold + START_DEAD_TIME_FUDGE;
420         tmg_set_delay(save_delay_line);
421         tmgptr->backplane_reset = 1;
422         tmg_set_test_mode(0);
423         return returncode;
424     }
425     else if((part >= 6) && (part < 30)) /* measure edge ramp slopes */
426     {
427         edge = part & 4;
428         aramp = (part - 6) / 4;
429         save_delay_line = tmg_get_delay();
430         tmg_set_delay(14);
431         if(tmg_measure_ramp(aramp, edge, &tmpcal, &errors) != SUCCESS)
432             returncode = FAILURE;
433         else
434             tmg_set_delay(save_delay_line);
435         tmg_set_test_mode(0);
436         return returncode;
437     }
438     else if(part == 30) /* check or set delay line */
439     {
440         save_delay_line = tmg_get_delay();
441         if(tmg_calibrate_delay(&tmpcal) != SUCCESS)
442             returncode = FAILURE;
443         if(!destructive)
444         {
445             if(abs((int)tmg_get_delay() - save_delay_line) > 1)
446             {
447                 returncode = FAILURE;
448                 tmg_set_delay(save_delay_line);
449             }
450             tmg_set_test_mode(0);
451             return returncode;
452         }
453         else if((part > 30) && (part < 45)) /* measure edge ramp offsets */
454         {
455             edge = (part - 37) & 4;
456             aramp = (part - 37) / 4;
457             if(tmg_measure_offset(aramp, edge, &tmpcal) != SUCCESS)
458                 returncode = FAILURE;
459             else
460                 tmg_set_test_mode(0);
461             return returncode;
462         }
463         else if((part == 45) && !destructive) /* measure Edge 7 min thresh */
464         {
465             if(tmg_measure_edge7_sample_minth(01, 01, &thresh7, &thresholds) != SUCCESS)
466                 returncode = FAILURE;
467             for(aramp = 0; aramp < NUMBER_OF_ERAMPS; aramp++)
468                 tmpcal.Edge7MinThreshold[aramp] = thresh7 + START_DEAD_TIME_FUDGE;
469             for(aramp = 0; aramp < NUMBER_OF_ERAMPS; aramp++)
470                 tmpcal.SampleMinThreshold[aramp] = thresholds + START_DEAD_TIME_FUDGE;
471             tmg_set_test_mode(0);
472             return returncode;
473         }
474         else if((part > 45) && (part < 50)) /* measure Edge 7 slope */
475         {
476             aramp = (part - 46);
477             if(tmg_measure_edge7_ramp(aramp, &tmpcal) != SUCCESS)
478                 returncode = FAILURE;
479         }
480     }

```


Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_cal.c

DATE 5/23/89

PAGE #

TIME 4:41:30 pm

5/152

```

LINE # SOURCE TEXT
481 tmg_set_test_mode(0);
482 return Returncode;
483 }
484 else if((part >= 50) && (part < 54)) /* measure sample slope */
485 {
486     aramp = (part - 50);
487     switch(aramp)
488     {
489         case 0: aramp = 0; break;
490         case 1: aramp = 1; break;
491         case 2: aramp = 1; break;
492         case 3: aramp = 2; break;
493     }
494     if(tmg_measure_sample_ramp(aramp, aramp, &stampcal, &errors) != SUCCESS)
495         Returncode = FAILURE;
496     tmg_set_test_mode(0);
497     return Returncode;
498 }
499 else if((part >= 54) && (part < 70))
500 {
501     aramp = (part - 54) % 4;
502     aramp = (part - 54) / 4;
503     if(tmg_measure_sample_offset_daly(aramp, aramp, &stampcal,
504         EDGEWISELTHRESHOLD) != SUCCESS)
505         Returncode = FAILURE;
506     tmg_set_test_mode(0);
507     return Returncode;
508 }
509 else if(part == 70)
510     return(tmg_process_and_transfer_calib_structure(&stampcal, &scalib));
511 else
512     return FAILURE;
513 }
514
515 tmg_calibrate_delay(cal)
516 register struct CALIB *cal;
517 {
518     static char me[] = "tmg_calibrate_delay";
519     long delta;
520     register long i;
521     register long longest_delay_edge, count;
522
523     lm_message("Setting delay line");
524
525     /* Now find the ramp 0 edge with the longest delay */
526     /* where delay = edgeoffset + (edgeminthresh * edgeoffset) */
527     longest_delay_edge = tmg_longest_delay_edge(01, cal);
528
529     /* Set the aramp delay */
530     if (tmg_compute_delay(01, longest_delay_edge, cal, &delta)
531         != SUCCESS) {
532         tmg_report_failure(me, "tmg_compute_delay");
533         return FAILURE;
534     }
535     if(tmg_cal_debug & 0x10000)
536         lm_message("as: adjusting delay by %d ps\n", me, delta);
537     tmg_adjust_delay(delta); /* do coarse adjustment */
538     if (tmg_compute_delay(01, longest_delay_edge, cal, &delta)
539         != SUCCESS) {
540         tmg_report_failure(me, "tmg_compute_delay");
541         return FAILURE;
542     }
543     for (count = 0, i = (delta > 0), i += (delta < 0); i != count) {
544         if (delta == 0)
545             break;
546         delta = (delta > 0) ? 1000 : -1000;
547         tmg_adjust_delay(delta); /* do fine adjustment */
548         if (tmg_compute_delay(01, longest_delay_edge, cal, &delta)
549             != SUCCESS) {
550             tmg_report_failure(me, "tmg_compute_delay");
551             return FAILURE;
552         }
553         if (count > 20) {
554             tmg_report_failure(me, "Delay adjustment");
555             return FAILURE;
556         }
557     }
558     lm_message("Done\n");
559     return SUCCESS;
560 }
561
562 /* tmg_assume_minth() assumes that the aramp 0 minths have been measured
563 ** and that it is a good assumption that the other ramps minths are the
564 ** same
565 */
566 static void
567 tmg_assume_minth(cal)
568 register struct CALIB *cal;
569 {
570     register long threshold0, edge, ramp;
571
572     for (edge = 0; edge < NUMBER_OF_EDGES; ++edge) {
573         threshold0 = cal->EdgeMinThresh[0][edge];
574         for (ramp = 1; ramp < NUMBER_OF_RAMPS; ++ramp) {
575             /* set min thresh of ramp to ramp 0's value */
576             cal->EdgeMinThresh[ramp][edge] = threshold0;
577         }
578     }
579 }
580
581 /* tmg_longest_delay_edge(ramp, cal)
582 ** returns the edge with the longest delay for a given ramp
583 ** where delay = offset + (minthresh * slope)
584 */
585 static long
586 tmg_longest_delay_edge(ramp, cal)
587 register long ramp;
588 register struct CALIB *cal;
589 {
590     register long edge, longest_delay_edge = 0;
591     register long delay, longest_delay;
592 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_cal.c

DATE 5/23/89

PAGE #

TIME 4:41:30 pm

6/153

```

LINE # SOURCE TEXT
601 longest_delay = cal->EdgeOffset[ramp][0]
602 + (long)(cal->EdgeMinThresh[ramp][0]
603 + cal->EdgeSlope[ramp][0]);
604 for (edge = 1; edge < NUMBER_OF_EDGES; ++edge) {
605     delay = cal->EdgeOffset[ramp][edge]
606     + (long)(cal->EdgeMinThresh[ramp][edge]
607     + cal->EdgeSlope[ramp][edge]);
608     if (delay > longest_delay) {
609         longest_delay = delay;
610         longest_delay_edge = edge;
611     }
612 }
613 return longest_delay_edge;
614 }
615
616 struct measure_points {
617     long low, mid, high;
618     long slope; /* (psec/th) */
619 }measure_points[] = {
620     {104, 155, 250, 500},
621     {18, 135, 250, 2000},
622     {12, 125, 250, 10000},
623     {10, 125, 250, 50000},
624 };
625
626 struct measure_sample_points {
627     long low, mid, high;
628     long slope; /* (psec/th) */
629 }measure_sample_points[] = {
630     {10, 125, 250, 500},
631     {10, 125, 250, 1000},
632     {10, 125, 250, 2000},
633     {10, 125, 250, 4000},
634 };
635 /* long measure_early_sample_points[] = {104, 54, 29, 16}; */
636 long measure_early_sample_points[] = {125, 45, 37, 22};
637
638 struct measure_edge7_points {
639     long low, mid, high;
640     long slope; /* (psec/th) */
641 }measure_edge7_points[] = {
642     {104, 155, 250, 500},
643     {18, 135, 250, 2000},
644     {12, 125, 250, 10000},
645     {10, 125, 250, 50000},
646 };
647
648 /* t is threshold array, p is period array, for general use */
649 /* during calibration of all ramps */
650 static long p[NUMBER_OF_RAMP][NUMBER_OF_EDGES + 1][7];
651 static long t[NUMBER_OF_RAMP][NUMBER_OF_EDGES + 1][7];
652
653 tmg_measure_ramp(ramp, edge, cal, errors)
654 register long ramp, edge, *errors;
655 register struct CALIB *cal;
656 {
657     static char me[] = "tmg_measure_ramp";
658     long try_tl;
659     register long dt, dp, slope;
660     long xpoints, point, temperrors;
661     long *plist, *tlist, *xpoints;
662     register long count;
663
664     if (DIAGS
665         if (tmg_cal_debug & 2)
666             lm_message("tmg_measure_ramp(%d,%d,%x)\n", ramp, edge, cal);
667     #endif DIAGS
668     /*
669     ** 1. Find a point above but near the minimum threshold and longer
670     ** than the magic ship dead time.
671     ** 2. Find a point high on the ramp.
672     ** 3. Calculate the slope and offset of that line.
673     ** 3a. Use the line to predict several points to do a fine calibration.
674     ** 3b. Results of fine calibration are used for slope and offset.
675     ** 4. Find a point approx midway between the first two points using
676     ** results of 3b.
677     ** 5. Compute a linearity value.
678     **
679     ** Slope will be stored as a long. The units are v/s (or uv/us).
680     */
681     temperrors = 0;
682     tmg_set_slot_count(2);
683
684     t[ramp][edge][0] = measure_points[ramp].low;
685     if (!p[ramp][edge][0] = tmg_find_period(ramp, edge,
686         (long)t[ramp][edge][0]))
687     {
688         tmg_report_failure(me,
689             "can't find low period: tmg_find_period");
690         return(FAILURE);
691     }
692
693     t[ramp][edge][2] = measure_points[ramp].high;
694     if (!p[ramp][edge][2] = tmg_find_period(ramp, edge,
695         (long)t[ramp][edge][2]))
696     {
697         tmg_report_failure(me,
698             "can't find high period: tmg_find_period");
699         return(FAILURE);
700     }
701
702     if ((dt = t[ramp][edge][2] - t[ramp][edge][0]) == 0) {
703         lm_error("infinite edge ramp slope detected\n");
704         return FAILURE;
705     }
706
707     if ((dp = p[ramp][edge][2] - p[ramp][edge][0]) == 0) {
708         lm_error("zero edge ramp slope detected\n");
709         return FAILURE;
710     }
711
712     /* slopes are normally 500, 2000, 10000, and 50000 */
713     slope = (dp/dt);
714     if ((slope < 50) || (slope > 50000)) {
715         lm_error("Illegal edge ramp slope %d ps/thr\n", slope);
716         return FAILURE;
717     }
718     cal->EdgeSlope[ramp][edge] = slope;
719     cal->EdgeOffset[ramp][edge] = (long)p[ramp][edge][0] -
720

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_cal.c | DATE 5/23/89 | PAGE # 7/154 |
|--|--|--|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 721 | | (long)(cal->Edgeslope[ramp][edge]) * (long)t[ramp][edge][0], | | |
| 722 | | if(ramp == 0) | | |
| 723 | | npoints = 3; | | |
| 724 | | else | | |
| 725 | | npoints = 7; | | |
| 726 | | tmg_set_slot_count(npoints + 1); | | |
| 727 | | p[ramp][edge][0] = tmg_predict_period(Measure_points[ramp].high, | | |
| 728 | | cal->Edgeslope[ramp][edge], | | |
| 729 | | cal->EdgeOffset[ramp][edge])/npoints; | | |
| 730 | | try_t1 = tmg_predict_threshold(p[ramp][edge][0], | | |
| 731 | | cal->Edgeslope[ramp][edge], | | |
| 732 | | cal->EdgeOffset[ramp][edge]); | | |
| 733 | | count = 0; | | |
| 734 | | do { | | |
| 735 | | t[ramp][edge][0] = try_t1--; | | |
| 736 | | if (++count > 10) { | | |
| 737 | | tmg_report_failure(me, | | |
| 738 | | "Unusual ramp thresholds: tmg_find_period"); | | |
| 739 | | return(FAILURE); | | |
| 740 | | if ((p[ramp][edge][0] = tmg_find_period(ramp, edge, | | |
| 741 | | (long)t[ramp][edge][0])) | | |
| 742 | | { | | |
| 743 | | tmg_report_failure(me, | | |
| 744 | | "can't find new low period: tmg_find_period"); | | |
| 745 | | return(FAILURE); | | |
| 746 | | t[ramp][edge][npoints-1] = tmg_predict_threshold(p[ramp][edge][0] * npoints, | | |
| 747 | | cal->Edgeslope[ramp][edge], | | |
| 748 | | cal->EdgeOffset[ramp][edge]); | | |
| 749 | | } while ((t[ramp][edge][npoints-1] > 251); | | |
| 750 | | if ((p[ramp][edge][npoints-1] = tmg_find_precise_period(ramp, edge, | | |
| 751 | | (long)t[ramp][edge][npoints-1], (long)p[ramp][edge][0], npoints))) | | |
| 752 | | { | | |
| 753 | | tmg_report_failure(me, | | |
| 754 | | "can't find high period: tmg_find_precise_period"); | | |
| 755 | | tmg_set_slot_count(2); | | |
| 756 | | return(FAILURE); | | |
| 757 | | /* do rest of points */ | | |
| 758 | | for(point = 1; point < npoints - 1; point++) | | |
| 759 | | { | | |
| 760 | | t[ramp][edge][point] = tmg_predict_threshold(p[ramp][edge][0] * (point+1), | | |
| 761 | | cal->Edgeslope[ramp][edge], | | |
| 762 | | cal->EdgeOffset[ramp][edge]); | | |
| 763 | | if ((p[ramp][edge][point] = tmg_find_precise_period(ramp, edge, | | |
| 764 | | (long)t[ramp][edge][point], (long)p[ramp][edge][0], point + 1))) | | |
| 765 | | { | | |
| 766 | | tmg_report_failure(me, | | |
| 767 | | "can't find mid period: tmg_find_precise_period"); | | |
| 768 | | tmg_set_slot_count(2); | | |
| 769 | | return(FAILURE); | | |
| 770 | | tmg_set_slot_count(2); | | |
| 771 | | if(ramp == 0) | | |
| 772 | | { | | |
| 773 | | plist = p[ramp][edge]; | | |
| 774 | | tlist = t[ramp][edge]; | | |
| 775 | | noodpts = npoints; | | |
| 776 | | } else | | |
| 777 | | { | | |
| 778 | | plist = t(p[ramp][edge][2]); | | |
| 779 | | tlist = t(t[ramp][edge][2]); | | |
| 780 | | noodpts = npoints - 2; | | |
| 781 | | } temperrors = (calculate_slope_and_offset_and_linearity(ramp,edge,cal, | | |
| 782 | | tlist,plist,noodpts,0); | | |
| 783 | | == SUCCESS ? 0 : 1; | | |
| 784 | | *errors += temperrors; | | |
| 785 | | return (temperrors ? FAILURE : SUCCESS); | | |
| 786 | | tmg_measure_offset(ramp, edge, cal) | | |
| 787 | | register long ramp, edge; | | |
| 788 | | register struct CALIS *cal; | | |
| 789 | | { | | |
| 790 | | static char me[] = "tmg_measure_offset"; | | |
| 791 | | long period; | | |
| 792 | | long threshold; | | |
| 793 | | long offset; | | |
| 794 | | in_message("-"); | | |
| 795 | | #ifdef DIAGS | | |
| 796 | | if (tmg_cal_debug & 2) | | |
| 797 | | in_message("tmg_measure_offset(td,td,tz)\n", ramp, edge, cal); | | |
| 798 | | #endif DIAGS | | |
| 799 | | /* | | |
| 800 | | .. 1. Find a point near the high end of the ramp. | | |
| 801 | | .. 2. Measure delay of that point. | | |
| 802 | | .. 3. Using known slope, calculate the offset of that line. | | |
| 803 | | */ | | |
| 804 | | tmg_set_slot_count(2); | | |
| 805 | | threshold = 250; | | |
| 806 | | if ((period = tmg_find_period(ramp, edge, threshold))) { | | |
| 807 | | tmg_report_failure(me, | | |
| 808 | | "can't find low period: tmg_find_period"); | | |
| 809 | | return(FAILURE); | | |
| 810 | | cal->EdgeOffset[ramp][edge] = tmg_predict_offset(period, threshold, | | |
| 811 | | cal->Edgeslope[ramp][edge]); | | |
| 812 | | if (tmg_cal_debug & 8) | | |
| 813 | | in_message("ramp td edge td offset is td ps (td p td t td s)\n", | | |
| 814 | | ramp, edge, cal->EdgeOffset[ramp][edge], | | |
| 815 | | period, threshold, cal->Edgeslope[ramp][edge]); | | |
| 816 | | return SUCCESS; | | |
| 817 | | calculate_slope_and_offset_and_linearity(ramp,edge,cal,t,p,npoints,ramp) | | |
| 818 | | register long ramp, edge; | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_cal.c

DATE 5/23/89
TIME 4:41:30 pm
PAGE # 8/155

```

LINE # SOURCE TEXT
841 long t[]; p[], numpoints, aramp;
842 struct CALIS "cal;
843 {
844     static char me[] = "calculate_slope_and_offset_and_linearity";
845     void fit();
846     float midpredicted, error, foffset, fslope;
847     long period;
848     long slope; /* picosec/threshold */
849     long offset; /* picoseconds */
850     long linearity; /* millipercent */
851     register int i;
852     long slope_delta, ideal_slope;
853     int nonlinear;
854
855     float x[7], y[7], siga, sigb, chl2, q, "xx, "yy;
856 #ifdef LOG_DEFINED
857     float sigy[7], "asigy;
858     long p, junk;
859     asigy = sigy + 1;
860 #endif LOG_DEFINED
861
862     xx = x - 1;
863     yy = y - 1;
864     for(i = 0; i < numpoints; i++)
865     {
866         x[i] = (float) t[i];
867         y[i] = (float) p[i];
868     }
869
870 #ifdef DIAGS
871     if (tmg_cal_debug & 0x1000000)
872     {
873         for(i = 0; i < numpoints; i++)
874             lm_message("x[%d] = %f ", i, x[i]);
875         lm_message("\n");
876         for(i = 0; i < numpoints; i++)
877             lm_message("y[%d] = %f ", i, y[i]);
878         lm_message("\n");
879     }
880 #endif DIAGS
881
882 #ifdef LOG_DEFINED
883     for(i = 0; i < numpoints; i++)
884     {
885         if (tmg_get_next_lower_period(p[i],
886             &p, &junk, &junk, &junk) != SUCCESS) {
887             tmg_report_failure(me, "tmg_get_frequency_setting(1)");
888             return FAILURE;
889         }
890         sigy[i] = (p[i] - p);
891     }
892
893     fit(xx, yy, numpoints, asigy, 1, &foffset, &fslope, &siga, &sigb, &chl2, &q);
894     linearity = (q * 1000.0);
895 #else LOG_DEFINED
896     fit(xx, yy, numpoints, 0, 0, &foffset, &fslope, &siga, &sigb, &chl2, &q);
897 #endif LOG_DEFINED
898
899     slope = fslope;
900     offset = foffset;
901
902     midpredicted = foffset + (float)t[numpoints >> 1] * fslope;
903     miderror = fabs(midpredicted - (float)p[numpoints >> 1]);
904     linearity = (long)((100000 + miderror));
905
906     if (tmg_cal_debug & 0x1000000) {
907         lm_message("slope = %d offset = %d linearity = %d numpoints = %d\n",
908             slope, offset, linearity, numpoints);
909     }
910
911     switch(edge) {
912     case 0:
913     case 1:
914     case 2:
915     case 3:
916     case 4:
917     case 5:
918         cal->EdgeOffset[ramp][edge] = offset;
919         cal->EdgeSlope[ramp][edge] = slope;
920         cal->EdgeLinearity[ramp][edge] = linearity;
921         ideal_slope = Measure_points[ramp].slope;
922         break;
923     case 6: /* really edge 7 */
924         cal->Edge7Slope[ramp] = slope;
925         cal->Edge7Linearity[ramp] = linearity;
926         ideal_slope = Measure_edge7_points[ramp].slope;
927         break;
928     case 7: /* really sample */
929         cal->SampleOffset[ramp][aramp] = offset;
930         cal->SampleSlope[ramp] = slope;
931         cal->SampleLinearity[ramp] = linearity;
932         ideal_slope = Measure_sample_points[aramp].slope;
933         break;
934     default:
935         lm_error("%s: Illegal edge %d\n", me, edge);
936         break;
937     }
938
939     nonlinear = (linearity > 2500); /* 2500mt == 2.3% */
940     if (nonlinear) {
941         lm_error("Ramp %d Edge %d slope %d failed linearity (%dmt)\n",
942             ramp, edge, slope, linearity);
943     }
944
945 #ifdef DIAGS
946     if ((tmg_cal_debug & 0x2000000) || nonlinear) {
947         lm_message("calculate_slope_and_offset_and_linearity(%d,%d):\n",
948             ramp, edge);
949         for(i = 0; i < numpoints; i++)
950         {
951             lm_message("t[%d] = %d p[%d] = %d f[%d] = %f x[%d] = %f y[%d] = %f\n",
952                 i, t[i], i, p[i], i, f[i], i, x[i], i, y[i]);
953         }
954         lm_message("fslope = %d slope = %d\n", fslope, slope);
955         lm_message("foffset = %d offset = %d\n", foffset, offset);
956         lm_message("midpredicted = %f linearity = %dmt\n",
957             midpredicted, linearity);
958         lm_message("miderror = %f\n", miderror);
959         lm_message("period = %d\n", period);
960     }

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_cal.c | DATE 5/23/89 | PAGE # 9/156 |
|--|--|-----------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 961 | in_message("threshold = %d\n", | | | |
| 962 | tmg_predict_threshold(period, slope, offset)); | | | |
| 963 | } | | | |
| 964 | #endif DIAGS | | | |
| 965 | if (scalib == 0) { | | | |
| 966 | return USE; | | | |
| 967 | } | | | |
| 968 | slope_delta = ideal_slope - 0.15; /* 15% */ | | | |
| 969 | if ((slope < (ideal_slope - slope_delta)) | | | |
| 970 | ((slope > (ideal_slope + slope_delta))) { | | | |
| 971 | in_error("Ramp %d edge %d slope %d out of range (%d +/- 15%)\n", | | | |
| 972 | (edge == 0)? ramp: ramp, edge, slope, ideal_slope); | | | |
| 973 | return FAILURE; | | | |
| 974 | } | | | |
| 975 | else | | | |
| 976 | return SUCCESS; | | | |
| 977 | } | | | |
| 978 | /* | | | |
| 979 | tmg_set_period(ramp, period); | | | |
| 980 | /* sets the clock to the closest period >= PERIOD psec and waits | | | |
| 981 | /* for the pll to settle | | | |
| 982 | */ | | | |
| 983 | #define CLOCK_SET_TIMEOUT 10000 | | | |
| 984 | tmg_set_period(ramp, period); | | | |
| 985 | long ramp, *period; | | | |
| 986 | { | | | |
| 987 | long timeout; | | | |
| 988 | static char me[] = "tmg_set_period"; | | | |
| 989 | long jitter, a, k, select; | | | |
| 990 | long actual; | | | |
| 991 | if (tmg_get_frequency_setting("period, | | | |
| 992 | actual, &jitter, &a, &k, &select) != SUCCESS) { | | | |
| 993 | tmg_report_failure(me, "tmg_get_frequency_setting"); | | | |
| 994 | return FAILURE; | | | |
| 995 | } | | | |
| 996 | #endif DIAGS | | | |
| 997 | if (tmg_cal_debug & 0x08) { | | | |
| 998 | in_message("%s: period %d actual %d psec a = %d k = %d sel = %d\n", | | | |
| 999 | me, "period, actual, a, k, select); | | | |
| 1000 | #endif DIAGS | | | |
| 1001 | period = actual; | | | |
| 1002 | if (!tmg_set_frequency((u_char)a, (u_char)k, (u_char)select) != SUCCESS) | | | |
| 1003 | { | | | |
| 1004 | tmg_report_failure(me, "tmg_set_frequency"); | | | |
| 1005 | return FAILURE; | | | |
| 1006 | } | | | |
| 1007 | /* Set the sample pulse width so the pcc doesn't break */ | | | |
| 1008 | /* Do this while the pll settles */ | | | |
| 1009 | tmg_set_sample_width(actual, measure_points[ramp].slope, | | | |
| 1010 | (long)TMC_MIN_SAMPLE_WIDTH); | | | |
| 1011 | for (timeout = CLOCK_SET_TIMEOUT; | | | |
| 1012 | timeout && (!tmg_check_locked()) != SUCCESS; | | | |
| 1013 | --timeout) | | | |
| 1014 | { | | | |
| 1015 | in_error("%s: CLOCK SET TIMED OUT\n", me); | | | |
| 1016 | return FAILURE; | | | |
| 1017 | } | | | |
| 1018 | return SUCCESS; | | | |
| 1019 | } | | | |
| 1020 | tmg_set_sample_width(period, slope, pw) | | | |
| 1021 | long period, slope, pw; | | | |
| 1022 | { | | | |
| 1023 | register long sample_start_time, sample_width; | | | |
| 1024 | if (pw < TMC_MIN_SAMPLE_WIDTH) return FAILURE; | | | |
| 1025 | sample_start_time = 2 * 512 * slope; | | | |
| 1026 | sample_width = (sample_start_time + pw)/period; | | | |
| 1027 | if (sample_width > 255) | | | |
| 1028 | sample_width = 255; | | | |
| 1029 | if (tmg_cal_debug & 0x00000008) | | | |
| 1030 | in_message("period = %d, sample width = %d (%d pw)\n", | | | |
| 1031 | period, sample_width, sample_width * period); | | | |
| 1032 | tmg_set_sample_width_reg(sample_width); | | | |
| 1033 | return SUCCESS; | | | |
| 1034 | } | | | |
| 1035 | #define MAX_THRESHOLD 50 /* 250 mv */ | | | |
| 1036 | tmg_measure_minth(ramp, edge, threshold) | | | |
| 1037 | long ramp, edge, *threshold; | | | |
| 1038 | { | | | |
| 1039 | static char me[] = "tmg_measure_minth"; | | | |
| 1040 | long period = 100175; /* 9.21 MHz == 100.375 ns */ | | | |
| 1041 | long found_high; | | | |
| 1042 | if (tmg_clockoff() != SUCCESS) { | | | |
| 1043 | in_error("%s: find_min_th: could not turn clock off\n"); | | | |
| 1044 | return FAILURE; | | | |
| 1045 | } | | | |
| 1046 | tmg_set_test_mode(1); tmg_set_slot_count(1); | | | |
| 1047 | for (*threshold = 4; *threshold <= MAX_THRESHOLD; ++*threshold) { | | | |
| 1048 | if (tmg_detect_always_high(ramp, edge, *period, | | | |
| 1049 | *threshold, 100001, &found_high) != SUCCESS) { | | | |
| 1050 | tmg_report_failure(me, "tmg_detect_always_high"); | | | |
| 1051 | return FAILURE; | | | |
| 1052 | } | | | |
| 1053 | if (found_high) { | | | |
| 1054 | break; | | | |
| 1055 | } | | | |
| 1056 | if (*threshold > MAX_THRESHOLD) { | | | |
| 1057 | in_error("%s: testmode 1: no min threshold for edge %d below %d\n", | | | |
| 1058 | me, edge, MAX_THRESHOLD); | | | |
| 1059 | return FAILURE; | | | |
| 1060 | } | | | |
| 1061 | if (tmg_cal_debug & 8) | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/trng_cal.c | DATE 5/23/89 | PAGE # 10/157 |
|--|--|---|-----------------|------------------|
| LINE # | | SOURCE TEXT | | |
| 1081 | | lm_message("Ramp td edge td min threshold = td\n", ramp, edge, "threshold"); | | |
| 1082 | | #ifdef DIAGS | | |
| 1083 | | if (trng_cal_debug & 1) | | |
| 1084 | | trng_play_til_key(); | | |
| 1085 | | #endif DIAGS | | |
| 1086 | | /* Now do double ramp trng_set_test_mode(2) test */ | | |
| 1087 | | trng_set_test_mode(2); trng_set_slot_count(1); | | |
| 1088 | | for(threshold = MAX_THRESHOLD; --threshold;) | | |
| 1089 | | if (trng_detect_always_low(ramp, edge, &period, | | |
| 1090 | | threshold, 100001, &found_high) != SUCCESS) { | | |
| 1091 | | trng_report_failure(me, "trng_detect_always_low"); | | |
| 1092 | | trng_set_test_mode(1); trng_set_slot_count(2); | | |
| 1093 | | return FAILURE; | | |
| 1094 | | } | | |
| 1095 | | if (found_high) { | | |
| 1096 | | break; | | |
| 1097 | | } | | |
| 1098 | | if (threshold > MAX_THRESHOLD) { | | |
| 1099 | | lm_error("sta: no min threshold for edge td below td\n", | | |
| 1100 | | me, edge, MAX_THRESHOLD); | | |
| 1101 | | trng_set_test_mode(1); trng_set_slot_count(2); | | |
| 1102 | | return FAILURE; | | |
| 1103 | | } | | |
| 1104 | | if (trng_cal_debug & 8) | | |
| 1105 | | lm_message("Ramp td edge td min threshold = td after testmode 2\n", ramp, edge, "threshold"); | | |
| 1106 | | #ifdef DIAGS | | |
| 1107 | | if (trng_cal_debug & 1) | | |
| 1108 | | trng_play_til_key(); | | |
| 1109 | | #endif DIAGS | | |
| 1110 | | trng_set_test_mode(1); trng_set_slot_count(2); | | |
| 1111 | | return SUCCESS; | | |
| 1112 | | } | | |
| 1113 | | trng_measure_edge7_sample_minth(aramp, aramp, thresh7, threshs) | | |
| 1114 | | long eramp, aramp, "thresh7, "threshs; | | |
| 1115 | | { | | |
| 1116 | | static char me[] = "trng_measure_edge7_sample_minth"; | | |
| 1117 | | long period = 100000; | | |
| 1118 | | long found_high; | | |
| 1119 | | if (trng_clockoff() != SUCCESS) { | | |
| 1120 | | lm_error("sta: trng_measure_edge7_sample_minth: could not turn clock off\n", | | |
| 1121 | | return FAILURE; | | |
| 1122 | | } | | |
| 1123 | | for(thresh7 = 4; thresh7 <= MAX_THRESHOLD; ++thresh7) { | | |
| 1124 | | lm_message(" "); | | |
| 1125 | | if (trng_detect_sample_always_high(aramp, aramp, &period, "thresh7, 101, | | |
| 1126 | | 100001, &found_high) != SUCCESS) { | | |
| 1127 | | trng_report_failure(me, "trng_detect_sample_always_high"); | | |
| 1128 | | goto cleanup; | | |
| 1129 | | } | | |
| 1130 | | if (found_high) | | |
| 1131 | | break; | | |
| 1132 | | #ifdef DIAGS | | |
| 1133 | | if (trng_cal_debug & 0x100) | | |
| 1134 | | trng_play_til_key(); | | |
| 1135 | | #endif DIAGS | | |
| 1136 | | if (thresh7 > MAX_THRESHOLD) { | | |
| 1137 | | lm_error("sta: no min threshold for edge 7 below td\n", | | |
| 1138 | | me, MAX_THRESHOLD); | | |
| 1139 | | goto cleanup; | | |
| 1140 | | } | | |
| 1141 | | /* lm_message("Ramp td edge 7 min threshold = td\n", aramp, "thresh7"); */ | | |
| 1142 | | for(threshs = 4; threshs <= MAX_THRESHOLD; ++threshs) { | | |
| 1143 | | lm_message(" "); | | |
| 1144 | | if (trng_detect_sample_always_high(aramp, aramp, &period, 101, "threshs, | | |
| 1145 | | 100001, &found_high) != SUCCESS) { | | |
| 1146 | | trng_report_failure(me, "trng_detect_sample_always_high"); | | |
| 1147 | | goto cleanup; | | |
| 1148 | | } | | |
| 1149 | | if (found_high) | | |
| 1150 | | break; | | |
| 1151 | | #ifdef DIAGS | | |
| 1152 | | if (trng_cal_debug & 0x100) | | |
| 1153 | | trng_play_til_key(); | | |
| 1154 | | #endif DIAGS | | |
| 1155 | | if (threshs > MAX_THRESHOLD) { | | |
| 1156 | | lm_error("sta: no min threshold for Sample below td\n", | | |
| 1157 | | me, MAX_THRESHOLD); | | |
| 1158 | | goto cleanup; | | |
| 1159 | | } | | |
| 1160 | | /* lm_message("Ramp td edge 7 min threshold = td\n", 0, "threshs"); */ | | |
| 1161 | | return SUCCESS; | | |
| 1162 | | } | | |
| 1163 | | cleanup: | | |
| 1164 | | return FAILURE; | | |
| 1165 | | } | | |
| 1166 | | trng_detect_sample(eramp, aramp, mode, period, thresh7, threshs, count, actual) | | |
| 1167 | | register long eramp, aramp; | | |
| 1168 | | long *period, *actual, thresh7, threshs, count; | | |
| 1169 | | int mode; | | |
| 1170 | | { | | |
| 1171 | | static char me[] = "trng_detect_sample"; | | |
| 1172 | | trng_set_eramp(eramp); | | |
| 1173 | | trng_set_aramp(aramp); | | |
| 1174 | | trng_set_edge_7_threshold(thresh7); | | |
| 1175 | | trng_set_sample_threshold(threshs); | | |
| 1176 | | if (trng_set_period(eramp, period) != SUCCESS) { | | |
| 1177 | | } | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_cal.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:30 pm | 11/158 |

```

1201 tmg_report_failure(me, "tmg_set_period");
1202 return FAILURE;
1203 }
1204
1205 #ifdef DIAGS
1206     if (tmg_cal_debug & 0x100) {
1207         lm_message("tmg_detect_sample: td psec\n", "period");
1208     }
1209 #endif DIAGS
1210
1211     if (tmg_detect(0l, count, actual, mode) != SUCCESS) {
1212         tmg_report_failure(me, "tmg_detect");
1213         return FAILURE;
1214     }
1215 #ifdef DIAGS
1216     if (tmg_cal_debug & 0x100)
1217         lm_message("tmg_detect_sample: count = %d\n", "actual");
1218 #endif DIAGS
1219     return SUCCESS;
1220 }
1221
1222 /*
1223  ** tmg_detect_always_low() is a magic function that reads the edge register
1224  ** up to "count" times and returns 1 if the value is always 0.
1225  ** Is our application remember that a sometimes high
1226  ** is as good as an always high, so this function returns 0 upon finding
1227  ** the first high.
1228  **
1229  ** In searching for the threshold period, we will search for the shortest
1230  ** period that sometimes goes high.
1231  **
1232  ** This is the edge we will seek.
1233  */
1234
1235 tmg_detect_edge(ramp, edge, mode, period, threshold, count, actual);
1236 register long ramp, edge;
1237 register long *period, *actual;
1238 long threshold, count;
1239 {
1240     static char me[] = "tmg_detect_edge";
1241     tmg_set_xramp(ramp);
1242     tmg_set_threshold(edge, threshold);
1243     if (tmg_set_period(ramp, period) != SUCCESS) {
1244         tmg_report_failure(me, "tmg_set_period");
1245         return FAILURE;
1246     }
1247 #ifdef DIAGS
1248     if (tmg_cal_debug & 2) {
1249         lm_message("tmg_detect_edge: td psec\n", "period");
1250     }
1251 #endif DIAGS
1252     if (tmg_detect(edge, count, actual, mode) != SUCCESS) {
1253         tmg_report_failure(me, "tmg_detect");
1254         return FAILURE;
1255     }
1256 #ifdef DIAGS
1257     if (tmg_cal_debug & 8)
1258         lm_message("tmg_detect_edge: count = %d\n", "actual");
1259 #endif DIAGS
1260     return SUCCESS;
1261 }
1262
1263 int tmg_detect(edge, repcount, count, mode)
1264 register long edge, repcount, *count;
1265 int mode;
1266 {
1267     register long mask;
1268     static char me[] = "tmg_detect";
1269     register long previous = 0, this;
1270     *count = 0;
1271     if (tmg_essiclear() != SUCCESS) {
1272         lm_error("tmg_essiclear: cal flip flops did not clear\n");
1273         tmg_report_failure(me, "tmg_essiclear");
1274         return FAILURE;
1275     }
1276     mask = 1 << edge;
1277     if (tmg_cal_debug & 2)
1278         printf("tmg_detect(td,td,td,td)\n", edge, repcount, count, mode);
1279     while (repcount-- > 0) {
1280         if (tmg_play((long) TIMEOUT) != SUCCESS) {
1281             tmg_report_failure(me, "tmg_play");
1282             return FAILURE;
1283         }
1284         if (mode & DETECT_SAMPLE)
1285             this = (tmg_get_sample_cal());
1286         else
1287             this = (tmg_get_edge_cal() & mask);
1288         if ((this == previous) != ((mode & DETECT_LOW) != 0)) {
1289             if (mode & DETECT_BOOL) {
1290                 *count = 0;
1291             }
1292             #ifdef DIAGS
1293             if (tmg_cal_debug & 4)
1294                 lm_message("td: repcount = %d\n",
1295                     me, repcount);
1296             #endif DIAGS
1297             return SUCCESS; /* Found a wrong */
1298         } else {
1299             ++*count;
1300             previous = this;
1301         }
1302     }
1303     if (mode & DETECT_BOOL) {
1304         *count = (*count != 0);
1305     }
1306     return SUCCESS; /* Found all rights */
1307 }
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_cal.c

DATE 5/23/89

PAGE #

TIME 4:41:30 pm

12/159

```

1321 #define SEARCH_COUNT 100
1322
1323 tmg_find_period(ramp, edge, thr)
1324 long ramp, edge, thr;
1325 {
1326     static char me[] = "tmg_find_period";
1327     long period, count;
1328     register long upper_bound, lower_bound;
1329
1330 #ifdef info
1331     1. Select the first period such that for the fastest possible
1332        ramp or the slowest possible ramp, the ramp discharges
1333        during the inactive phase of the first cycle (Set period
1334        to 1/3 longer than expected ramp speed);
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440

```


| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_cal.c | DATE 5/23/89 | PAGE # 13/160 |
|--|--|-----------------------------------|-----------------|------------------|
| LINE # | SOURCE TEXT | | | |
| 1441 | if (tmg_cal_debug & 4) | | | |
| 1442 | lm_message("(got it between td and td)\n", | | | |
| 1443 | lower_bound, upper_bound); | | | |
| 1444 | #endif DIAGS | | | |
| 1445 | return period; | | | |
| 1446 | } | | | |
| 1447 | if (!found_high) { | | | |
| 1448 | /* too short - raise lower bound */ | | | |
| 1449 | lower_bound = period; | | | |
| 1450 | } else { | | | |
| 1451 | /* too long - lower upper bound */ | | | |
| 1452 | upper_bound = period; | | | |
| 1453 | } | | | |
| 1454 | } | | | |
| 1455 | lm_error("BINARY EDGE SEARCH FAILED TO CONVERGE!\n"); | | | |
| 1456 | return period; | | | |
| 1457 | } | | | |
| 1458 | } | | | |
| 1459 | tmg_find_precise_period(ramp, edge, thr, base_period, multiplier) | | | |
| 1460 | long ramp, edge, thr, base_period, multiplier; | | | |
| 1461 | { | | | |
| 1462 | static char me[] = "tmg_find_precise_period"; | | | |
| 1463 | long period, count1, count2; | | | |
| 1464 | register long upper_bound, lower_bound, delta; | | | |
| 1465 | } | | | |
| 1466 | #ifdef info | | | |
| 1467 | 1. Select the first period based on the predicted base value. | | | |
| 1468 | 2. Compute delta based on the period and multiplier. | | | |
| 1469 | 3. If you detect low, decrease period by delta. If you still | | | |
| 1470 | detect low, then error out. | | | |
| 1471 | 4. If you detect high, increase period by delta. If you still | | | |
| 1472 | detect high, then error out. | | | |
| 1473 | 5. Do a binary search for the exact low-to-high period. | | | |
| 1474 | #endif info | | | |
| 1475 | period = base_period; | | | |
| 1476 | #ifdef DIAGS | | | |
| 1477 | if (tmg_cal_debug & 0x2) | | | |
| 1478 | lm_message("tmg_find_precise_period(td,td,td,td)\n", | | | |
| 1479 | ramp, edge, thr, base_period, multiplier); | | | |
| 1480 | } | | | |
| 1481 | if (tmg_cal_debug & 4) | | | |
| 1482 | lm_message("ts: starting at td psec\n", me, period); | | | |
| 1483 | #endif DIAGS | | | |
| 1484 | delta = (period / 4) / multiplier; | | | |
| 1485 | { | | | |
| 1486 | if (tmg_detect_always_low(ramp, edge, | | | |
| 1487 | period, thr, (long)SEARCH_COUNT, &count1) != SUCCESS) { | | | |
| 1488 | tmg_report_failure(me, "tmg_detect_always_low(1)"); | | | |
| 1489 | return 0; | | | |
| 1490 | } | | | |
| 1491 | #ifdef DIAGS | | | |
| 1492 | if (tmg_cal_debug & 1) | | | |
| 1493 | tmg_play_til_key(); | | | |
| 1494 | #endif DIAGS | | | |
| 1495 | if (!count1) { | | | |
| 1496 | lower_bound = period; | | | |
| 1497 | period += delta; | | | |
| 1498 | } else { | | | |
| 1499 | upper_bound = period; | | | |
| 1500 | period -= delta; | | | |
| 1501 | } | | | |
| 1502 | } | | | |
| 1503 | } | | | |
| 1504 | { | | | |
| 1505 | if (tmg_detect_always_low(ramp, edge, | | | |
| 1506 | period, thr, (long)SEARCH_COUNT, &count2) != SUCCESS) { | | | |
| 1507 | tmg_report_failure(me, "tmg_detect_always_low(2)"); | | | |
| 1508 | return 0; | | | |
| 1509 | } | | | |
| 1510 | #ifdef DIAGS | | | |
| 1511 | if (tmg_cal_debug & 1) | | | |
| 1512 | tmg_play_til_key(); | | | |
| 1513 | #endif DIAGS | | | |
| 1514 | if (!count2) { | | | |
| 1515 | lower_bound = period; | | | |
| 1516 | } else { | | | |
| 1517 | upper_bound = period; | | | |
| 1518 | } | | | |
| 1519 | } | | | |
| 1520 | } | | | |
| 1521 | if (count1 == count2) { | | | |
| 1522 | lm_error("ts: ts on ramp td edge td\n", me, | | | |
| 1523 | "Can't find edge boundaries", ramp, edge); | | | |
| 1524 | return 0; | | | |
| 1525 | } | | | |
| 1526 | return tmg_edge_search(ramp, edge, thr, upper_bound, lower_bound) | | | |
| 1527 | * multiplier; | | | |
| 1528 | } | | | |
| 1529 | } | | | |
| 1530 | tmg_sample_search(aramp, aramp, thr7, thr, upper_bound, lower_bound) | | | |
| 1531 | long aramp, aramp, thr7, thr, upper_bound, lower_bound; | | | |
| 1532 | { | | | |
| 1533 | static char me[] = "tmg_sample_search"; | | | |
| 1534 | long period, found_high; | | | |
| 1535 | long timeout = BIN_LIMIT; | | | |
| 1536 | for (timeout = 0; timeout <= BIN_LIMIT; ++timeout) { | | | |
| 1537 | period = (upper_bound + lower_bound)/2; | | | |
| 1538 | if (tmg_detect_sample_always_low(aramp, aramp, (long*)&period, | | | |
| 1539 | thr7, thr, (long)SEARCH_COUNT, (long*)&found_high) | | | |
| 1540 | != SUCCESS) | | | |
| 1541 | { | | | |
| 1542 | tmg_report_failure(me, | | | |
| 1543 | "tmg_detect_sample_always_low(3)"); | | | |
| 1544 | } | | | |
| 1545 | #ifdef DIAGS | | | |
| 1546 | if (tmg_cal_debug & 0x100) | | | |
| 1547 | tmg_play_til_key(); | | | |
| 1548 | #endif DIAGS | | | |
| 1549 | if ((period <= lower_bound) (period >= upper_bound)) { | | | |
| 1550 | /* We can't get more exact than this */ | | | |
| 1551 | #ifdef DIAGS | | | |
| 1552 | if (tmg_cal_debug & 0x400) | | | |
| 1553 | lm_message("(got it between td and td)\n", | | | |
| 1554 | lower_bound, upper_bound); | | | |
| 1555 | #endif DIAGS | | | |
| 1556 | return period; | | | |
| 1557 | } | | | |
| 1558 | } | | | |
| 1559 | if (!found_high) { | | | |
| 1560 | /* too short - raise lower bound */ | | | |

| Copyright 1989 Logic Modeling Systems | SOURCE PROGRAM diags/tmg_cal.c | DATE 5/23/89 TIME 4:41:30 pm | PAGE # 14/161 |
|--|--|---------------------------------------|------------------|
| LINE # | SOURCE TEXT | | |
| 1561 | lower_bound = period; | | |
| 1562 |) else { | | |
| 1563 | /* too long - lower upper bound */ | | |
| 1564 | upper_bound = period; | | |
| 1565 | } | | |
| 1566 | } | | |
| 1567 | lm_error("PRIMARY SAMPLE SEARCH FAILED TO CONVERGE!\n"); | | |
| 1568 | return period; | | |
| 1569 | } | | |
| 1570 | /* tmg_sample_search */ | | |
| 1571 | tmg_find_precise_sample_period(arm, arm, thr7, thr, base_period, multiplier) | | |
| 1572 | long arm, arm, thr7, thr, base_period, multiplier; | | |
| 1573 | { | | |
| 1574 | static char me[] = "tmg_find_precise_sample_period"; | | |
| 1575 | long period, count1, count2; | | |
| 1576 | register long upper_bound, lower_bound, delta; | | |
| 1577 | } | | |
| 1578 | if (info) | | |
| 1579 | 1. Select the first period based on the predicted base value. | | |
| 1580 | 2. Compute delta based on the period and multiplier. | | |
| 1581 | 3. If you detect low, decrease period by delta. If you still | | |
| 1582 | detect low, then error out. | | |
| 1583 | 4. If you detect high, increase period by delta. If you still | | |
| 1584 | detect high, then error out. | | |
| 1585 | 5. Do a binary search for the exact low-to-high period. | | |
| 1586 | endif info | | |
| 1587 | period = base_period; | | |
| 1588 | if (info) | | |
| 1589 | if (tmg_cal_debug & 0x200) | | |
| 1590 | lm_message("tmg_find_precise_sample_period(%d,%d,%d,%d,%d,%d)\n", | | |
| 1591 | arm, arm, thr7, thr, base_period, multiplier); | | |
| 1592 | if (tmg_cal_debug & 0x1000) | | |
| 1593 | lm_message("ts: starting at %d psec\n", me, period); | | |
| 1594 | endif DIAGS | | |
| 1595 | delta = (period / 4) / multiplier; | | |
| 1596 | { | | |
| 1597 | if (tmg_detect_sample_always_low(arm, arm, | | |
| 1598 | period, thr7, thr, (long)SEARCH_COUNT, &count1) != SUCCESS) { | | |
| 1599 | tmg_report_failure(me, | | |
| 1600 | "tmg_detect_sample_always_low(1)"); | | |
| 1601 | return 0; | | |
| 1602 | } | | |
| 1603 | if (tmg_cal_debug & 0x100) | | |
| 1604 | tmg_play_til_key(); | | |
| 1605 | endif DIAGS | | |
| 1606 | if (!count1) { | | |
| 1607 | lower_bound = period; | | |
| 1608 | period -= delta; | | |
| 1609 | } else { | | |
| 1610 | upper_bound = period; | | |
| 1611 | period += delta; | | |
| 1612 | } | | |
| 1613 | { | | |
| 1614 | if (tmg_detect_sample_always_low(arm, arm, | | |
| 1615 | period, thr7, thr, (long)SEARCH_COUNT, &count2) != SUCCESS) { | | |
| 1616 | tmg_report_failure(me, | | |
| 1617 | "tmg_detect_sample_always_low(2)"); | | |
| 1618 | return 0; | | |
| 1619 | } | | |
| 1620 | if (tmg_cal_debug & 0x100) | | |
| 1621 | tmg_play_til_key(); | | |
| 1622 | endif DIAGS | | |
| 1623 | if (!count2) { | | |
| 1624 | lower_bound = period; | | |
| 1625 | } else { | | |
| 1626 | upper_bound = period; | | |
| 1627 | } | | |
| 1628 | } | | |
| 1629 | if (count1 == count2) { | | |
| 1630 | lm_error("ts: ts on edge ramp to sample ramp to\n", me, | | |
| 1631 | "Can't find sample boundaries", arm, arm); | | |
| 1632 | return 0; | | |
| 1633 | } | | |
| 1634 | return tmg_sample_search(arm, arm, thr7, thr, upper_bound, lower_bound) | | |
| 1635 | * multiplier; | | |
| 1636 | } | | |
| 1637 | /* tmg_find_precise_sample_period */ | | |
| 1638 | tmg_find_sample_period(arm, arm, thrash7, thrash) | | |
| 1639 | long arm, arm, thrash7, thrash; | | |
| 1640 | { | | |
| 1641 | static char me[] = "tmg_find_sample_period"; | | |
| 1642 | long found_high, period, count, | | |
| 1643 | register long upper_bound, lower_bound, | | |
| 1644 | long timeout; | | |
| 1645 | if (info) | | |
| 1646 | 1. Select the first period such that for the fastest possible | | |
| 1647 | ramp or the slowest possible ramp, the ramp discharges | | |
| 1648 | during the inactive phase of the first cycle (Set period | | |
| 1649 | to 1/3 longer than expected Ramp speed); | | |
| 1650 | endif info | | |
| 1651 | XXXXXXX | | |
| 1652 | XXXXXXX | | |
| 1653 | XXXXXXX | | |
| 1654 | XXXXXXX | | |
| 1655 | XXXXXXX | | |
| 1656 | XXXXXXX | | |
| 1657 | XXXXXXX | | |
| 1658 | XXXXXXX | | |
| 1659 | XXXXXXX | | |
| 1660 | XXXXXXX | | |
| 1661 | XXXXXXX | | |
| 1662 | XXXXXXX | | |
| 1663 | XXXXXXX | | |
| 1664 | XXXXXXX | | |
| 1665 | XXXXXXX | | |
| 1666 | XXXXXXX | | |
| 1667 | XXXXXXX | | |
| 1668 | XXXXXXX | | |
| 1669 | XXXXXXX | | |
| 1670 | XXXXXXX | | |
| 1671 | XXXXXXX | | |
| 1672 | XXXXXXX | | |
| 1673 | XXXXXXX | | |
| 1674 | XXXXXXX | | |
| 1675 | XXXXXXX | | |
| 1676 | XXXXXXX | | |
| 1677 | XXXXXXX | | |
| 1678 | XXXXXXX | | |
| 1679 | XXXXXXX | | |
| 1680 | XXXXXXX | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_cal.c | DATE 5/23/89 TIME 4:41:30 pm | PAGE # 15/162 |
|--|---|-----------------------------------|---------------------------------------|------------------|
| LINE # | SOURCE TEXT | | | |
| 1681 | 2. If you detect low, go to step 3 (see point B in the figure). | | | |
| 1682 | Otherwise (see point A in the figure), keep reducing | | | |
| 1683 | the period by .5 until you detect a low. | | | |
| 1684 | 3. Reduce the frequency by .5 until you get a high. Remember the last | | | |
| 1685 | frequencies that you got high and low. | | | |
| 1686 | 4. Do a binary search for the low-to-high period. | | | |
| 1687 | Sendif info | | | |
| 1688 | if(threshs == 10) /* must be looking at edge? ramp */ | | | |
| 1689 | upper_bound = (4 * Measure_edge_points[aramp].slope * 255)/3; | | | |
| 1690 | else | | | |
| 1691 | upper_bound = (4 * Measure_sample_points[aramp].slope * 255)/3 + 40000; | | | |
| 1692 | /* if (upper_bound < MIN_PERIOD) then something is wrong */ | | | |
| 1693 | #endif DIAGS | | | |
| 1694 | if (tmg_cal_debug & 0x1000) | | | |
| 1695 | lm_message("tmg_find_sample_period: starting at %d msec\n", | | | |
| 1696 | upper_bound); | | | |
| 1697 | #endif DIAGS | | | |
| 1698 | /* correct for first time through loop: */ | | | |
| 1699 | period = (100 * upper_bound)/tmg_reduction_factor; | | | |
| 1700 | do { | | | |
| 1701 | upper_bound = period; | | | |
| 1702 | period = (tmg_reduction_factor * period)/100; | | | |
| 1703 | if (period < MIN_PERIOD) | | | |
| 1704 | period = MIN_PERIOD; | | | |
| 1705 | if (tmg_detect_sample_always_low(aramp, aramp, | | | |
| 1706 | period, threshs, (long)SEARCH_COUNT, &count) != SUCCESS) { | | | |
| 1707 | tmg_report_failure(me, "tmg_detect_sample_always_low(1)"); | | | |
| 1708 | return 0; | | | |
| 1709 | #endif DIAGS | | | |
| 1710 | if (tmg_cal_debug & 0x100) | | | |
| 1711 | tmg_play_till_key(); | | | |
| 1712 | #endif DIAGS | | | |
| 1713 | if (period == upper_bound) { | | | |
| 1714 | lm_error("tmg_find_sample_period: Could not find low\n"); | | | |
| 1715 | return 0; | | | |
| 1716 | } while (!count); | | | |
| 1717 | #endif DIAGS | | | |
| 1718 | if (tmg_cal_debug & 0x100) | | | |
| 1719 | lm_message("found low at %d\n", period); | | | |
| 1720 | #endif DIAGS | | | |
| 1721 | do { | | | |
| 1722 | upper_bound = period; | | | |
| 1723 | period = (tmg_reduction_factor * period)/100; | | | |
| 1724 | if (period < MIN_PERIOD) | | | |
| 1725 | period = MIN_PERIOD; | | | |
| 1726 | if (tmg_detect_sample_always_low(aramp, aramp, | | | |
| 1727 | period, threshs, (long)SEARCH_COUNT, &count) != SUCCESS) { | | | |
| 1728 | tmg_report_failure(me, "tmg_detect_sample_always_low(2)"); | | | |
| 1729 | return 0; | | | |
| 1730 | #endif DIAGS | | | |
| 1731 | if (tmg_cal_debug & 0x100) | | | |
| 1732 | tmg_play_till_key(); | | | |
| 1733 | #endif DIAGS | | | |
| 1734 | if (period == upper_bound) { | | | |
| 1735 | lm_error("tmg_find_sample_period: Could not find 2nd high\n"); | | | |
| 1736 | return 0; | | | |
| 1737 | } while (!count); | | | |
| 1738 | #endif DIAGS | | | |
| 1739 | if (tmg_cal_debug & 0x100) | | | |
| 1740 | lm_message("found second high at %d\n", period); | | | |
| 1741 | #endif DIAGS | | | |
| 1742 | lower_bound = period; | | | |
| 1743 | for (timeout = 0; timeout <= BIN_LIMIT; ++timeout) { | | | |
| 1744 | period = (upper_bound + lower_bound)/2; | | | |
| 1745 | if (tmg_detect_sample_always_low(aramp, aramp, period, | | | |
| 1746 | threshs, (long)SEARCH_COUNT, &found_high) != SUCCESS) { | | | |
| 1747 | tmg_report_failure(me, "tmg_detect_sample_always_low(3)"); | | | |
| 1748 | #endif DIAGS | | | |
| 1749 | if (tmg_cal_debug & 0x100) | | | |
| 1750 | tmg_play_till_key(); | | | |
| 1751 | #endif DIAGS | | | |
| 1752 | if ((period <= lower_bound) (period >= upper_bound)) { | | | |
| 1753 | /* We can't get more exact than this */ | | | |
| 1754 | #endif DIAGS | | | |
| 1755 | if (tmg_cal_debug & 0x100) | | | |
| 1756 | lm_message("got it between %d and %d\n", | | | |
| 1757 | lower_bound, upper_bound); | | | |
| 1758 | #endif DIAGS | | | |
| 1759 | return period; | | | |
| 1760 | } if (!found_high) { | | | |
| 1761 | /* too short - raise lower bound */ | | | |
| 1762 | lower_bound = period; | | | |
| 1763 | } else { | | | |
| 1764 | /* too long - lower upper bound */ | | | |
| 1765 | upper_bound = period; | | | |
| 1766 | } | | | |
| 1767 | lm_error("BINARY SAMPLE EDGE SEARCH FAILED TO CONVERGE!\n"); | | | |
| 1768 | return period; | | | |
| 1769 | #endif | | | |
| 1770 | tmg_compute_delay(ramp, edge, cal, delay) | | | |
| 1771 | register long ramp, edge; | | | |
| 1772 | register long *delay; | | | |
| 1773 | register struct CALIB *cal; | | | |
| 1774 | { | | | |
| 1775 | static char me[] = "tmg_compute_delay"; | | | |
| 1776 | long adjust; | | | |
| 1777 | /* We assume that this ramp has already been measured */ | | | |
| 1778 | #endif | | | |
| 1779 | if (tmg_measure_offset(ramp, edge, cal) != SUCCESS) { | | | |
| 1780 | tmg_report_failure(me, "tmg_measure_offset"); | | | |
| 1781 | return FAILURE; | | | |
| 1782 | } | | | |
| 1783 | adjust = MAGIC_DEAD_TIME | | | |
| 1784 | - cal->EdgeOffset[ramp][edge]; | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_cal.c | DATE 5/23/89 | PAGE # 16/163 |
|--|---|-----------------------------------|-----------------|------------------|
| LINE # | SOURCE TEXT | | | |
| 1801 | - (long)(cal->Edgeslope[ramp][edge] - cal->EdgesInThresh[ramp][edge]); | | | |
| 1802 | | | | |
| 1803 | #ifdef DIAGS | | | |
| 1804 | if (tmg_cal_debug & 0x0000) { | | | |
| 1805 | lm_message("Give an offset of td psec, slope of td psec/thresh.\n", | | | |
| 1806 | cal->EdgesInThresh[ramp][edge], cal->Edgeslope[ramp][edge]); | | | |
| 1807 | lm_message("a threshold of td and an estimated dead time of td psec.\n", | | | |
| 1808 | cal->EdgesInThresh[ramp][edge], MAGIC_DEAD_TIME); | | | |
| 1809 | lm_message("I think we should adjust the delay by td psec.\n", | | | |
| 1810 | adjust); | | | |
| 1811 | } | | | |
| 1812 | #endif DIAGS | | | |
| 1813 | *delay = MAGIC_DEAD_TIME - cal->EdgesOffset[ramp][edge] | | | |
| 1814 | - cal->Edgeslope[ramp][edge] * cal->EdgesInThresh[ramp][edge]; | | | |
| 1815 | return(SUCCESS); | | | |
| 1816 | | | | |
| 1817 | | | | |
| 1818 | | | | |
| 1819 | void | | | |
| 1820 | tmg_adjust_delay(delta) | | | |
| 1821 | register long delta; /* microseconds */ | | | |
| 1822 | { | | | |
| 1823 | /* delay and delta are expressed in ns */ | | | |
| 1824 | long delay = tmg_get_delay(); | | | |
| 1825 | | | | |
| 1826 | #ifdef DIAGS | | | |
| 1827 | if (tmg_cal_debug & 0x0000) | | | |
| 1828 | lm_message("Present delay = td ns (td)\n", delay, | | | |
| 1829 | tmg_get_delay_reg()); | | | |
| 1830 | #endif DIAGS | | | |
| 1831 | delay += delta/1000; /* microseconds */ | | | |
| 1832 | | | | |
| 1833 | #ifdef DIAGS | | | |
| 1834 | if (tmg_cal_debug & 0x0000) | | | |
| 1835 | lm_message("Setting delay to td ns\n", delay); | | | |
| 1836 | #endif DIAGS | | | |
| 1837 | | | | |
| 1838 | /* should we do bounds checking here. bill? */ | | | |
| 1839 | tmg_set_delay(delay); | | | |
| 1840 | | | | |
| 1841 | #ifdef DIAGS | | | |
| 1842 | if (tmg_cal_debug & 0x0000) | | | |
| 1843 | lm_message("New delay = td ns (td)\n", tmg_get_delay(), | | | |
| 1844 | tmg_get_delay_reg()); | | | |
| 1845 | #endif DIAGS | | | |
| 1846 | | | | |
| 1847 | | | | |
| 1848 | /* | | | |
| 1849 | tmg_get_frequency_setting() | | | |
| 1850 | /* | | | |
| 1851 | This routine computes the values for n, k, and clock | | | |
| 1852 | select register setting to give the best approximation | | | |
| 1853 | of the desired clock period. The clock period provided | | | |
| 1854 | is always equal to or greater than that requested. | | | |
| 1855 | The actual clock period is related to n, k as follows: | | | |
| 1856 | period = (k / n) * T_REF, where | | | |
| 1857 | k = 1, 2, ..., 256 or k = 3, 4, ..., 512, | | | |
| 1858 | n = 128, 129, ..., 256, and | | | |
| 1859 | T_REF = 256 / 10 MHz = 256 reference period. | | | |
| 1860 | /* | | | |
| 1861 | Identical to tmg_get_frequency_setting without max bounds checking. | | | |
| 1862 | /* | | | |
| 1863 | Returns: SUCCESS except if period violates min bound. | | | |
| 1864 | */ | | | |
| 1865 | #define I_T_REF 253333 /* reference period in ps */ | | | |
| 1866 | #define K_CRITICAL 251 | | | |
| 1867 | { | | | |
| 1868 | tmg_get_frequency_setting(period,actualptr,jitterptr,nptr,kptr,selectptr) | | | |
| 1869 | register long period; /* physical clock period in ps */ | | | |
| 1870 | register long *actualptr; /* actual clock period in ps */ | | | |
| 1871 | register long *jitterptr; /* jitter per period in ps */ | | | |
| 1872 | register long *nptr; | | | |
| 1873 | register long *kptr; | | | |
| 1874 | register long *selectptr; | | | |
| 1875 | { | | | |
| 1876 | register long estimate, error, besterror; | | | |
| 1877 | register int k, n, save_k = 0, save_n = 0; | | | |
| 1878 | register long pef = period/0x0000, sf; | | | |
| 1879 | if ((period < 33333) (period > 3500000)) | | | |
| 1880 | return(FAILURE); | | | |
| 1881 | for(besterror = 0x7fffffff, n = N_MIN, n <= N_MAX, n++) { | | | |
| 1882 | for ((error = (n * pef)/512), sf = 0, error, error >= 1) { | | | |
| 1883 | ++sf; | | | |
| 1884 | /* use the best-approximate formula for this period... */ | | | |
| 1885 | k = (n * (period >> sf)) / (I_T_REF >> sf); | | | |
| 1886 | if (k > K_CRITICAL) | | | |
| 1887 | estimate = k * (I_T_REF/n); | | | |
| 1888 | else | | | |
| 1889 | estimate = (k * I_T_REF)/n; | | | |
| 1890 | if (estimate < period) | | | |
| 1891 | ++k; | | | |
| 1892 | if (k > 512) /* see if out of range */ | | | |
| 1893 | continue; /* if so, ignore it */ | | | |
| 1894 | if (k > 256) /* see if above 256 */ | | | |
| 1895 | if (k & 1) /* see if odd */ | | | |
| 1896 | k++; /* make even */ | | | |
| 1897 | if (k > K_CRITICAL) | | | |
| 1898 | estimate = k * (I_T_REF/n); | | | |
| 1899 | else | | | |
| 1900 | estimate = (k * I_T_REF)/n; | | | |
| 1901 | if ((error = estimate - period) < 0) | | | |
| 1902 | continue; | | | |
| 1903 | if (error < besterror) { | | | |
| 1904 | save_n = n; | | | |
| 1905 | save_k = k; | | | |
| 1906 | besterror = error; | | | |
| 1907 | } | | | |
| 1908 | if (besterror == 0x7fffffff) { | | | |
| 1909 | return(FAILURE); | | | |
| 1910 | } | | | |
| 1911 | *nptr = 256 - save_n + 1; /* value to put in reg */ | | | |
| 1912 | | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_cal.c

DATE 5/23/89
TIME 4:41:30 pm

PAGE #
17/164

```

LINE # SOURCE TEXT
1921 if (save_k == 1) {
1922     *kptr = 255; /* special case */
1923     *selectptr = 0; /* select div by 2 */
1924 } else {
1925     if (save_k > 256) {
1926         *kptr = 256 - (save_k >> 1) + 1; /* divide k by 2 */
1927         *selectptr = 2; /* select div by 4k */
1928     } else {
1929         *kptr = 256 - save_k + 1; /* normal case */
1930         *selectptr = 1; /* select div by 2k */
1931     }
1932 }
1933 /* actualptr = T_REF * save_k / save_n */
1934 if (save_k > K_CRITICAL) {
1935     *actualptr = save_k * (I_T_REF/save_n);
1936 } else {
1937     *actualptr = (save_k * I_T_REF)/save_n;
1938 }
1939 *jitterptr = 20 * 2 * save_k; /* 20 ps per PLL clock */
1940 return(SUCCESS);
1941 }
1942
1943 tmg_get_best_lower_period(period, actualptr, jitterptr, sptr, kptr, selectptr)
1944 register long period; /* physical clock period in ps */
1945 register long *actualptr; /* actual clock period in ps */
1946 register long *jitterptr; /* jitter per period in ps */
1947 register long *sptr;
1948 register long *kptr;
1949 register long *selectptr;
1950 {
1951     register long estimate, error, besterror;
1952     register int k, n, save_k = 0, save_n = 0;
1953     register long paf = period/0x10000, af;
1954
1955     if ((period < 33333) || (period > 35000000))
1956         return(FAILURE);
1957
1958     for (besterror = 0x7fffffff, n = N_MIN, k = N_MAX; n++ <= N_MAX; k--) {
1959         for ((error = (n * paf)/512), af = 0, error; error >= 1; ) {
1960             ++af;
1961             /* use the most accurate formula for this period... */
1962             k = (n * (period >> af)) / (I_T_REF >> af);
1963             if (k > K_CRITICAL)
1964                 estimate = k * (I_T_REF/n);
1965             else
1966                 estimate = (k * I_T_REF)/n;
1967             if (estimate >= period)
1968                 ++k;
1969             if (k > 512) /* see if out of range */
1970                 continue; /* if so, ignore it */
1971             if (k > 256) /* see if above 256 */
1972                 if (k & 1) /* see if odd */
1973                     k++; /* make even */
1974             if (k > K_CRITICAL)
1975                 estimate = k * (I_T_REF/n);
1976             else
1977                 estimate = (k * I_T_REF)/n;
1978             if ((error = period - estimate) <= 0)
1979                 continue;
1980             if (error < besterror) {
1981                 save_n = n;
1982                 save_k = k;
1983                 besterror = error;
1984             }
1985         }
1986     }
1987     if (besterror == 0x7fffffff) {
1988         return(FAILURE);
1989     }
1990     *sptr = 256 - save_n + 1; /* value to put in reg */
1991     if (save_k == 1) {
1992         *kptr = 255; /* special case */
1993         *selectptr = 0; /* select div by 2 */
1994     } else {
1995         if (save_k > 256) {
1996             *kptr = 256 - (save_k >> 1) + 1; /* divide k by 2 */
1997             *selectptr = 2; /* select div by 4k */
1998         } else {
1999             *kptr = 256 - save_k + 1; /* normal case */
2000             *selectptr = 1; /* select div by 2k */
2001         }
2002     }
2003     /* actualptr = T_REF * save_k / save_n */
2004     if (save_k > K_CRITICAL) {
2005         *actualptr = save_k * (I_T_REF/save_n);
2006     } else {
2007         *actualptr = (save_k * I_T_REF)/save_n;
2008     }
2009     *jitterptr = 20 * 2 * save_k; /* 20 ps per PLL clock */
2010     return(SUCCESS);
2011 }
2012
2013 /* threshold and period array for sample calibration */
2014 static long st[NUMBER_OF_ERASES][NUMBER_OF_SHRAPS][7];
2015 static long sp[NUMBER_OF_ERASES][NUMBER_OF_SHRAPS][7];
2016
2017 tmg_measure_sample_ramp(erase, ramp, cal, errors)
2018 register long erase, ramp, *errors;
2019 register struct CALLS *cal;
2020 {
2021     static char me[] = "tmg_measure_sample_ramp";
2022     char buffer[64];
2023     long t17, t27;
2024     register long nopsists, slope, dp, dt, count, point, temperrors;
2025     long thresh, offset;
2026
2027 #ifdef DIAGS
2028     if (tmg_cal_debug & 0x200)
2029         lm_message("ts(td,tx)\n", me, ramp, cal);
2030 #endif
2031     /*
2032     ** 1. find a point above but near the minimum threshold and longer
2033     ** than the magic chip dead time.
2034     ** 2. find a point high on the ramp.
2035     ** 3. Calculate the slope and offset of that line.
2036     ** 3a. Use slope and offset to predict points for fine calibration.
2037     ** 3b. Results of fine calibration are used for slope and offset.
2038     ** 4. Find a point approx midway between the first two points
2039     ** using results of 3b.
2040     ** 5. Compute a linearity value.
2041     */

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_cal.c | DATE 5/23/89 | PAGE # 18/165 |
|--|---|-----------------------------------|-----------------|------------------|
| LINE # | SOURCE TEXT | | | |
| 2041 | /* | | | |
| 2042 | ** Slope will be stored as a long. The units are v/a (or uv/us). | | | |
| 2043 | */ | | | |
| 2044 | | | | |
| 2045 | temperrors = 0; | | | |
| 2046 | switch(aramp) | | | |
| 2047 | { | | | |
| 2048 | case 0: | | | |
| 2049 | case 1: | | | |
| 2050 | npoints = 3; | | | |
| 2051 | break; | | | |
| 2052 | case 2: | | | |
| 2053 | case 3: | | | |
| 2054 | npoints = 7; | | | |
| 2055 | break; | | | |
| 2056 | } | | | |
| 2057 | tmg_set_slot_count(2); | | | |
| 2058 | | | | |
| 2059 | tl7 = Measure_edge7_points[aramp].low; | | | |
| 2060 | st[aramp][aramp][0] = Measure_sample_points[aramp].low; | | | |
| 2061 | | | | |
| 2062 | if(! (sp[aramp][aramp][0] = tmg_find_sample_period(aramp, aramp, tl7, | | | |
| 2063 | (long)st[aramp][aramp][0])) | | | |
| 2064 | { | | | |
| 2065 | tmg_report_failure(me, | | | |
| 2066 | "can't find low period: tmg_find_sample_period"); | | | |
| 2067 | return(FAILURE); | | | |
| 2068 | } | | | |
| 2069 | | | | |
| 2070 | tl7 = Measure_edge7_points[aramp].low; | | | |
| 2071 | st[aramp][aramp][2] = Measure_sample_points[aramp].high; | | | |
| 2072 | if(! (sp[aramp][aramp][2] = tmg_find_sample_period(aramp, aramp, tl7, | | | |
| 2073 | (long)st[aramp][aramp][2])) | | | |
| 2074 | { | | | |
| 2075 | tmg_report_failure(me, | | | |
| 2076 | "can't find high period: tmg_find_sample_period"); | | | |
| 2077 | return(FAILURE); | | | |
| 2078 | } | | | |
| 2079 | | | | |
| 2080 | if ((dt = (st[aramp][aramp][2] - st[aramp][aramp][0]) == 0) { | | | |
| 2081 | in_error("Infinite sample ramp slope\n"); | | | |
| 2082 | return FAILURE; | | | |
| 2083 | } | | | |
| 2084 | | | | |
| 2085 | if ((dp = (sp[aramp][aramp][2] - sp[aramp][aramp][0]) == 0) { | | | |
| 2086 | in_error("Zero sample ramp slope\n"); | | | |
| 2087 | return FAILURE; | | | |
| 2088 | } | | | |
| 2089 | | | | |
| 2090 | /* slopes are normally 500, 1000, 2000, and 4000 */ | | | |
| 2091 | slope = dp/dt; | | | |
| 2092 | if ((slope < 50) (slope > 40000)) { | | | |
| 2093 | in_error("Illegal sample ramp slope to pa/thr\n", slope); | | | |
| 2094 | return FAILURE; | | | |
| 2095 | } | | | |
| 2096 | offset = sp[aramp][aramp][0] - slope*st[aramp][aramp][0]; | | | |
| 2097 | | | | |
| 2098 | sp[aramp][aramp][0] = tmg_predict_period(Measure_sample_points[aramp].high, | | | |
| 2099 | slope, offset) * 2 / (npoints + 1); | | | |
| 2100 | | | | |
| 2101 | thresh = tmg_predict_threshold((long)sp[aramp][aramp][0], | | | |
| 2102 | slope, offset); | | | |
| 2103 | | | | |
| 2104 | st[aramp][aramp][0] = thresh; | | | |
| 2105 | if(st[aramp][aramp][0] < cal->SampleMinThresh(aramp)) | | | |
| 2106 | { | | | |
| 2107 | errors++; | | | |
| 2108 | sprintf(buffer, "Can't fit to periods in sample ramp to", npoints, aramp); | | | |
| 2109 | tmg_report_failure(me, buffer); | | | |
| 2110 | return(FAILURE); | | | |
| 2111 | } | | | |
| 2112 | | | | |
| 2113 | count = 0; | | | |
| 2114 | do { | | | |
| 2115 | if (++count > 10) { | | | |
| 2116 | tmg_report_failure(me, | | | |
| 2117 | "Unusual ramp thresholds: tmg_find_sample_period"); | | | |
| 2118 | return(FAILURE); | | | |
| 2119 | } | | | |
| 2120 | if (! (sp[aramp][aramp][0] = tmg_find_sample_period(aramp, aramp, tl7, | | | |
| 2121 | (long)st[aramp][aramp][0])) | | | |
| 2122 | { | | | |
| 2123 | tmg_report_failure(me, | | | |
| 2124 | "can't find new low period: tmg_find_sample_period"); | | | |
| 2125 | return(FAILURE); | | | |
| 2126 | } | | | |
| 2127 | | | | |
| 2128 | st[aramp][aramp][npoints-1] = | | | |
| 2129 | tmg_predict_threshold(sp[aramp][aramp][0] * (npoints + 1) / 2, | | | |
| 2130 | slope, offset); | | | |
| 2131 | if(--tl7 < cal->Edge7MinThresh(aramp)) | | | |
| 2132 | { | | | |
| 2133 | tmg_report_failure(me, "No allowable setting for Edge 7 Threshold"); | | | |
| 2134 | return FAILURE; | | | |
| 2135 | } | | | |
| 2136 | } while (st[aramp][aramp][npoints-1] > 251); | | | |
| 2137 | | | | |
| 2138 | tmg_set_slot_count(npoints + 1); | | | |
| 2139 | if (! (sp[aramp][aramp][npoints-1] = | | | |
| 2140 | tmg_find_precise_sample_period(aramp, aramp, tl7, | | | |
| 2141 | (long)st[aramp][aramp][npoints-1], (long)(sp[aramp][aramp][0] | | | |
| 2142 | / 2), (long)(npoints + 1)))) | | | |
| 2143 | { | | | |
| 2144 | tmg_report_failure(me, | | | |
| 2145 | "can't find high period: tmg_find_precise_sample_period"); | | | |
| 2146 | tmg_set_slot_count(2); | | | |
| 2147 | return(FAILURE); | | | |
| 2148 | } | | | |
| 2149 | | | | |
| 2150 | /* do rest of points */ | | | |
| 2151 | for(point = 1; point < npoints - 1; point++) | | | |
| 2152 | { | | | |
| 2153 | st[aramp][aramp][point] = tmg_predict_threshold(sp[aramp][aramp][0] | | | |
| 2154 | * (point+2) / 2, slope, offset); | | | |
| 2155 | | | | |
| 2156 | if (! (sp[aramp][aramp][point] = | | | |
| 2157 | tmg_find_precise_sample_period(aramp, aramp, tl7, | | | |
| 2158 | (long)st[aramp][aramp][point], (long)(sp[aramp][aramp][0] | | | |
| 2159 | / 2), (long)(point + 2)))) | | | |
| 2160 | { | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_cal.c

DATE 5/23/89
TIME 4:41:30 pm

PAGE #
19/166

```

LINE #          SOURCE TEXT
2161      tmg_report_failure(me,
2162      "can't find mid period: tmg_find_precise_period");
2163      tmg_set_slot_count(2);
2164      return(FAILURE);
2165  }
2166  }
2167  tmg_set_slot_count(2);
2168  tmg_errors = (calculate_slope_and_offset_and_linearity(aramp, 71, cal,
2169  (long*)at(aramp)[aramp], (long*)ap(aramp)[aramp], soproints, aramp)
2170  == SUCCESS) ? 0 : 1;
2171  *errors += tmg_errors;
2172  return (tmg_errors ? FAILURE : SUCCESS);
2173  }
2174  }
2175  }
2176  tmg_measure_sample_offset_only(aramp, aramp, cal, mode)
2177  register long aramp, aramp;
2178  register struct CALIB *cal;
2179  int mode;
2180  {
2181      static char me[] = "tmg_measure_sample_offset_only";
2182      long period;
2183      long t7, ts;
2184      long threshold;
2185      long offset;
2186      //
2187      #ifdef DIAGS
2188      if (tmg_cal_debug & 0x100)
2189          lm_message("tmg_measure_sample_offset_only(td,td,tx,td)\n",
2190          aramp, aramp, cal, mode);
2191      #endif
2192      //
2193      // 1. if EDGE7SAMPLETRIGGERMODE, use a point on Edge7 about 40 ns out
2194      // else don't care about Edge 7
2195      // 2. if EDGE7SAMPLETRIGGERMODE, use a minimal point on Sample Ramp
2196      // else use point about 40 ns out
2197      // 3. Calculate offset using previously measured slopes
2198      // 4. Store results in calibration structure
2199      //
2200      if(mode == EDGE7SAMPLETRIGGERMODE)
2201      {
2202          t7 = Measure_edge7_points(aramp).low;
2203          ts = Measure_sample_points(aramp).low;
2204          if (!((period = tmg_find_sample_period(aramp, aramp, t7, ts))) {
2205              tmg_report_failure(me,
2206              "can't find low period: tmg_find_sample_period");
2207              return(FAILURE);
2208          }
2209          offset = (long)period
2210          - ((long)(cal->Edge7Slope[aramp] * t7)
2211          - (long)(cal->SampleSlope[aramp] * ts));
2212          threshold = (40000L - (long)offset
2213          - (long)(cal->Edge7Slope[aramp] * t7))
2214          / (long)(cal->SampleSlope[aramp]);
2215          if(tmg_cal_debug & 0x100)
2216              lm_message("predict offset = %ld, threshold = %ld\n",
2217              offset, threshold);
2218          if(threshold < cal->SampleMinThresh[aramp])
2219              threshold = cal->SampleMinThresh[aramp];
2220          if(threshold > 250)
2221              threshold = 255;
2222          ts = threshold;
2223          if (!((period = tmg_find_sample_period(aramp, aramp, t7, ts))) {
2224              tmg_report_failure(me,
2225              "can't find real low period: tmg_find_sample_period");
2226              return(FAILURE);
2227          }
2228          cal->SampleOffset[aramp][aramp] = (long)period
2229          - ((long)(cal->Edge7Slope[aramp] * t7)
2230          - (long)(cal->SampleSlope[aramp] * ts));
2231      }
2232      else if(mode == EARLYSAMPLETRIGGERMODE)
2233      {
2234          tmgptr->sample_mode = EARLYSAMPLETRIGGERMODE;
2235          t7 = 255; /* have to keep Edge 7 quiet! */
2236          ts = Measure_early_sample_points(aramp);
2237          if (!((period = tmg_find_sample_period(61, aramp, t7, ts))) {
2238              tmg_report_failure(me, "can't find low period: tmg_find_sample_period",
2239              early mode");
2240              tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE;
2241              return(FAILURE);
2242          }
2243          offset = (long)period
2244          - ((long)(cal->SampleSlope[aramp] * (long)ts
2245          - (long)(cal->SampleSlope[aramp] * (long)ts));
2246          if(threshold < cal->SampleMinThresh[aramp])
2247              threshold = cal->SampleMinThresh[aramp];
2248          if(threshold > 250)
2249              threshold = 255;
2250          ts = threshold;
2251          if (!((period = tmg_find_sample_period(61, aramp, t7, ts))) {
2252              tmg_report_failure(me, "can't find real low period: \
2253              tmg_find_sample_period, early mode");
2254              tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE;
2255              return(FAILURE);
2256          }
2257          cal->EarlySampleOffset[aramp] = (long)period
2258          - ((long)(cal->SampleSlope[aramp] * (long)ts
2259          - (long)(cal->SampleSlope[aramp] * (long)ts));
2260          tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE;
2261      }
2262      else
2263      {
2264          lm_error("Pussy mode passed to tmg_measure_sample_offset_only\n");
2265          return FAILURE;
2266      }
2267      return SUCCESS;
2268  }
2269  }
2270  }
2271  }
2272  }
2273  }
2274  }
2275  }
2276  }
2277  }
2278  }
2279  }
2280  }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_cal.c

DATE

5/23/89

PAGE #

TIME

4:41:30 pm

20/167

```

2281 tmg_measure_edge7_ramp(ramp, cal);
2282 register long ramp;
2283 register struct CALIB *cal;
2284
2285     static char me[] = "tmg_measure_edge7_ramp",
2286     long nopoints, count,
2287     int point,
2288     register long slope, offset, dt, dp;
2289     char buffer(80);
2290 #ifdef DIAGS
2291     if (tmg_cal_debug & 0x200)
2292         lm_message("tmg_measure_edge7_ramp(%d,%x)\n", ramp, cal);
2293 #endif
2294
2295     /* 1. find a point above but near the minimum threshold and longer
2296     /* than the magic chip dead time.
2297     /* 2. find a point high on the ramp.
2298     /* 3. Calculate the slope and offset of that line.
2299     /* 3a. Use slope and offset to predict points for fine calibration.
2300     /* 3b. Results of fine calibration are used for slope and offset.
2301     /* 4. Find a point approx midway between the first two points.
2302     /* 5. Compute a linearity value.
2303     /*
2304     /* Slope will be stored as a long. The units are v/s (or uv/us).
2305     */
2306
2307     if (ramp == 0)
2308         nopoints = 3;
2309     else
2310         nopoints = 7;
2311
2312     tmg_set_slot_count(2);
2313     t[ramp][6][0] = Measure_edge7_points(ramp).low;
2314     if (! (p[ramp][6][0] = tmg_find_sample_period(ramp, 01,
2315         (long)t[ramp][6][0], 101)))
2316     {
2317         tmg_report_failure(me,
2318             "can't find low period: tmg_find_sample_period");
2319         return (FAILURE);
2320     }
2321
2322     t[ramp][6][2] = Measure_edge7_points(ramp).high;
2323     if (! (p[ramp][6][2] = tmg_find_sample_period(ramp, 01,
2324         (long)t[ramp][6][2], 101)))
2325     {
2326         tmg_report_failure(me,
2327             "can't find high period: tmg_find_sample_period");
2328         return (FAILURE);
2329     }
2330
2331     if ((dt = t[ramp][6][2] - t[ramp][6][0]) == 0) {
2332         lm_error("Zero edge 7 ramp slope\n");
2333         return FAILURE;
2334     }
2335
2336     if ((dp = p[ramp][6][2] - p[ramp][6][0]) == 0) {
2337         lm_error("Zero edge 7 ramp slope\n");
2338         return FAILURE;
2339     }
2340
2341     /* slopes are normally 500, 2000, 10000, and 50000
2342     slope = dp/dt;
2343     if ((slope < 50) || (slope > 50000)) {
2344         lm_error("Illegal edge 7 ramp slope to pa/thr\n", slope);
2345         return FAILURE;
2346     }
2347     offset = p[ramp][6][0] - slope*t[ramp][6][0];
2348
2349     p[ramp][6][0] = tmg_predict_period(Measure_edge7_points(ramp).high,
2350         slope, offset) / nopoints;
2351     t[ramp][6][0] = tmg_predict_threshold(p[ramp][6][0], slope, offset);
2352     if (t[ramp][6][0] < cal->Edge7MinThresh(ramp))
2353     {
2354         sprintf(buffer, "Can't fit 2d periods in Edge7 ramp to", nopoints, ramp);
2355         tmg_report_failure(me, buffer);
2356         return (FAILURE);
2357     }
2358     count = 0;
2359     do {
2360         if (++count > 10) {
2361             tmg_report_failure(me,
2362                 "Unusual ramp thresholds: tmg_find_sample_period");
2363             return (FAILURE);
2364         }
2365         if (! (p[ramp][6][0] = tmg_find_sample_period(ramp, 01,
2366             (long)t[ramp][6][0], 101)))
2367         {
2368             tmg_report_failure(me,
2369                 "can't find low period: tmg_find_sample_period");
2370             return (FAILURE);
2371         }
2372
2373         t[ramp][6][nopoints-1] =
2374             tmg_predict_threshold(p[ramp][6][0] * nopoints,
2375                 slope, offset);
2376
2377         if (t[ramp][6][0] < cal->Edge7MinThresh(ramp))
2378         {
2379             tmg_report_failure(me, "No allowable setting for Edge 7 Threshold");
2380             return FAILURE;
2381         }
2382     } while (t[ramp][6][nopoints-1] > 251);
2383
2384     tmg_set_slot_count(nopoints + 1);
2385     if (! (p[ramp][6][nopoints-1] =
2386         tmg_find_precise_sample_period(ramp, 01,
2387             (long)t[ramp][6][nopoints-1], 101, (long)p[ramp][6][0],
2388             (long)nopoints)))
2389     {
2390         tmg_report_failure(me,
2391             "can't find high period: tmg_find_precise_sample_period");
2392         tmg_set_slot_count(2);
2393         return (FAILURE);
2394     }
2395
2396     /* do rest of points
2397     for (point = 1; point < nopoints - 1; point++)
2398     {
2399         t[ramp][6][point] = tmg_predict_threshold(p[ramp][6][0]
2400             * (point+1), slope, offset);

```


Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_cal.c

DATE

5/23/89

PAGE #

TIME

4:41:30 pm

21/168

```

LINE # SOURCE TEXT
2401
2402 if (!p(ramp)[6][point] =
2403     tmg_find_precise_sample_period(ramp, 01,
2404     (long)t(ramp)[6][point], 101, (long)p(ramp)[6][0],
2405     (long)(point - 1)))
2406 {
2407     tmg_report_failure(mw,
2408     "can't find mid period: tmg_find_precise_period"),
2409     tmg_set_slot_count(2),
2410     return(FAILURE);
2411 }
2412
2413 tmg_set_slot_count(2);
2414 return calculate_slope_and_offset_and_linearity(ramp, 61, cal,
2415     (long)t(ramp)[6], (long)p(ramp)[6], npoints, 01);
2416 }
2417
2418 tmg_complete_calib_structure(calptr)
2419 struct CALIB *calptr;
2420 {
2421     long eramp, aramp, min_edge_slope, max_edge_slope, ave_slope;
2422     long max_min_delay, min_max_delay;
2423     long edge;
2424     long temp;
2425
2426     /* Figure out EdgeMinDelay[] and EdgeMaxDelay[]
2427     /* average various edge slopes along with Edge7
2428
2429     eramp = 0;
2430     {
2431         max_edge_slope = 0;
2432         min_edge_slope = 1000000;
2433         max_min_delay = 0;
2434         min_max_delay = 100000000;
2435         ave_slope = calptr->Edge7Slope(eramp);
2436         for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2437         {
2438             if(calptr->EdgeSlope(eramp)[edge] > max_edge_slope)
2439                 max_edge_slope = calptr->EdgeSlope(eramp)[edge];
2440             if(calptr->EdgeSlope(eramp)[edge] < min_edge_slope)
2441                 min_edge_slope = calptr->EdgeSlope(eramp)[edge];
2442             ave_slope += calptr->EdgeSlope(eramp)[edge];
2443         }
2444         ave_slope /= NUMBER_OF_EDGES + 1;
2445         for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2446         {
2447             calptr->EdgeSlope(eramp)[edge] = ave_slope;
2448             /* min thresh + 2 for thermal tail effects
2449             temp = (long)(calptr->EdgeMinThresh(eramp)[edge] + 2) * max_edge_slope
2450             + calptr->EdgeOffset(eramp)[edge];
2451             if(max_min_delay < temp)
2452                 max_min_delay = temp;
2453             temp = (255 - END DEAD TIME FUDGE) * (long)min_edge_slope
2454             + calptr->EdgeOffset(eramp)[edge];
2455             - 2 * (long)ave_slope; /* jitter */
2456             if(min_max_delay > temp)
2457                 min_max_delay = temp;
2458         }
2459         calptr->Edge7Slope(eramp) = ave_slope;
2460         calptr->EdgeMinDelay(eramp) = max_min_delay;
2461         calptr->EdgeMaxDelay(eramp) = min_max_delay;
2462     }
2463     for(eramp = 1; eramp < NUMBER_OF_ERAMPS; eramp++)
2464     {
2465         max_edge_slope = 0;
2466         min_edge_slope = 1000000;
2467         max_min_delay = 0;
2468         min_max_delay = 100000000;
2469         ave_slope = calptr->Edge7Slope(eramp);
2470         for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2471         {
2472             if(calptr->EdgeSlope(eramp)[edge] > max_edge_slope)
2473                 max_edge_slope = calptr->EdgeSlope(eramp)[edge];
2474             if(calptr->EdgeSlope(eramp)[edge] < min_edge_slope)
2475                 min_edge_slope = calptr->EdgeSlope(eramp)[edge];
2476             ave_slope += calptr->EdgeSlope(eramp)[edge];
2477         }
2478         ave_slope /= NUMBER_OF_EDGES + 1;
2479         for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2480         {
2481             calptr->EdgeSlope(eramp)[edge] = ave_slope;
2482             /* min thresh + 2 for thermal tail effects
2483             temp = (long)(calptr->EdgeMinThresh(eramp)[edge] + 2) * max_edge_slope
2484             + calptr->EdgeOffset(eramp)[edge];
2485             if(max_min_delay < temp)
2486                 max_min_delay = temp;
2487             temp = (255 - END DEAD TIME FUDGE) * (long)min_edge_slope
2488             + calptr->EdgeOffset(eramp)[edge];
2489             - 2 * (long)ave_slope; /* jitter */
2490             if(min_max_delay > temp)
2491                 min_max_delay = temp;
2492         }
2493         calptr->Edge7Slope(eramp) = ave_slope;
2494         calptr->EdgeMinDelay(eramp) = max_min_delay;
2495         calptr->EdgeMaxDelay(eramp) = min_max_delay;
2496     }
2497     /* Now we have to account for cases where a small step in the ramp
2498     /* causes a large negative offset that fools us into thinking
2499     /* the minimum delay is small. The indication of this is when
2500     /* a slower ramp has a smaller minimum delay than a faster ramp
2501     if(calptr->EdgeMinDelay(eramp) < calptr->EdgeMinDelay(eramp-1))
2502     {
2503         for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2504         {
2505             /* increase the minimum threshold as necessary
2506             while(((long)(calptr->EdgeMinThresh(eramp)[edge] + 2)
2507             + calptr->EdgeSlope(eramp)[edge]
2508             + calptr->EdgeOffset(eramp)[edge])
2509             < calptr->EdgeMinDelay(eramp-1))
2510                 ++(calptr->EdgeMinThresh(eramp)[edge]);
2511         }
2512         /* recompute minimum delays
2513         for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2514         {
2515             temp = (long)(calptr->EdgeMinThresh(eramp)[edge] + 2) * max_edge_slope
2516             + calptr->EdgeOffset(eramp)[edge];
2517             if(max_min_delay < temp)
2518                 max_min_delay = temp;
2519         }
2520         calptr->EdgeMinDelay(eramp) = max_min_delay;
2521     }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_cal.c

DATE

5/23/89

PAGE #

TIME

4:41:30 pm

22/169

```

2521 }
2522
2523 /* now figure out SampleMinDelay */
2524 for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2525 {
2526     max_min_delay = 0;
2527     for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2528     {
2529         /* min thresh + 2 for thermal tail effects
2530          * calptr->Edge7MinThresh[aramp] + 2) */
2531         temp = (long)((calptr->Edge7MinThresh[aramp] + 2)
2532             * calptr->Edge7Slope[aramp])
2533             + (long)(calptr->SampleMinThresh[aramp] * calptr->SampleSlope[aramp])
2534             + calptr->SampleOffset[aramp];
2535         if(tmg_cal_debug & 0x10000000)
2536         {
2537             lm_message("aramp=%ld, aramp=%ld\n", aramp, aramp);
2538             lm_message("tEdge7MinTh = %ld, Edge7Slope = %ld\n",
2539                 calptr->Edge7MinThresh[aramp], calptr->Edge7Slope[aramp]);
2540             lm_message("tSampleMinTh = %ld, SampleSlope = %ld\n",
2541                 calptr->SampleMinThresh[aramp], calptr->SampleSlope[aramp]);
2542             lm_message("ttemp = %ld\n", temp);
2543         }
2544         if(max_min_delay < temp)
2545             max_min_delay = temp;
2546     }
2547     calptr->SampleMinDelay[aramp] = max_min_delay;
2548     if(tmg_cal_debug & 0x10000000)
2549         lm_message("tSampleMinDelay = %ld\n", max_min_delay);
2550 }
2551 for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2552 {
2553     calptr->EarlySampleMinDelay[aramp] = (long)(calptr->SampleMinThresh[aramp]
2554         * calptr->SampleSlope[aramp])
2555         + calptr->EarlySampleOffset[aramp];
2556     calptr->EarlySampleMaxDelay[aramp] = (2551 - END_DEAD_TIME_FUDGE)
2557         + (long)calptr->SampleSlope[aramp]
2558         + calptr->EarlySampleOffset[aramp];
2559 }
2560 return SUCCESS;
2561 }
2562
2563 /*
2564 * tmg_print_calib_structures(calptr)
2565 *
2566 * This routine prints the relevant stuff contained within
2567 * a calibration structure.
2568 *
2569 * Inputs: pointer to calib structure
2570 *
2571 * Outputs: Always returns SUCCESS
2572 */
2573 void tmg_print_calib_structures(calptr)
2574 struct CALIB *calptr;
2575 {
2576     long aramp, aramp;
2577     long edge;
2578     for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2579     {
2580         lm_message("Edge Ramp %d:\n", aramp);
2581         lm_message("tMinimum delay = %d ps\n", calptr->EdgeMinDelay[aramp]);
2582         lm_message("tMaximum delay = %d ps\n", calptr->EdgeMaxDelay[aramp]);
2583         lm_message("tSlope = %d ps/threshold\n", calptr->EdgeSlope[aramp][0]);
2584         lm_message("tEdge %t to %t %t %t %t %t %t %t %t\n",
2585             for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2586             lm_message("tEdge %t to %t %t %t %t %t %t %t %t\n",
2587                 lm_message("ttd", calptr->EdgeOffset[aramp][edge]);
2588                 lm_message("tlinearity");
2589                 for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2590                 lm_message("ttd", calptr->EdgeLinearity[aramp][edge]);
2591                 lm_message("ttd", calptr->EdgeLinearity[aramp][edge]);
2592             }
2593     }
2594     for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2595     {
2596         lm_message("tEdge Ramp %d:\n", aramp);
2597         lm_message("tSample Minimum Delay = %d\n", calptr->SampleMinDelay[aramp]);
2598         lm_message("tSample Ramp %t to %t %t %t %t %t %t %t %t\n",
2599             for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2600             lm_message("ttd", calptr->SampleOffset[aramp][aramp]);
2601         }
2602         lm_message("tSample Ramp %t to %t %t %t %t %t %t %t %t\n",
2603             for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2604             lm_message("ttd", calptr->SampleLinearity[aramp]);
2605             lm_message("ttd", calptr->SampleLinearity[aramp]);
2606             lm_message("ttd", calptr->SampleLinearity[aramp]);
2607             lm_message("ttd", calptr->SampleLinearity[aramp]);
2608             lm_message("ttd", calptr->SampleLinearity[aramp]);
2609             lm_message("ttd", calptr->SampleLinearity[aramp]);
2610             lm_message("ttd", calptr->SampleLinearity[aramp]);
2611             lm_message("ttd", calptr->SampleLinearity[aramp]);
2612             lm_message("ttd", calptr->SampleLinearity[aramp]);
2613             lm_message("ttd", calptr->SampleLinearity[aramp]);
2614             lm_message("ttd", calptr->SampleLinearity[aramp]);
2615             lm_message("ttd", calptr->SampleLinearity[aramp]);
2616             lm_message("ttd", calptr->SampleLinearity[aramp]);
2617             lm_message("ttd", calptr->SampleLinearity[aramp]);
2618             lm_message("ttd", calptr->SampleLinearity[aramp]);
2619             lm_message("ttd", calptr->SampleLinearity[aramp]);
2620             lm_message("ttd", calptr->SampleLinearity[aramp]);
2621             lm_message("ttd", calptr->SampleLinearity[aramp]);
2622             lm_message("ttd", calptr->SampleLinearity[aramp]);
2623             lm_message("ttd", calptr->SampleLinearity[aramp]);
2624             lm_message("ttd", calptr->SampleLinearity[aramp]);
2625             lm_message("ttd", calptr->SampleLinearity[aramp]);
2626             lm_message("ttd", calptr->SampleLinearity[aramp]);
2627             lm_message("ttd", calptr->SampleLinearity[aramp]);
2628             lm_message("ttd", calptr->SampleLinearity[aramp]);
2629             lm_message("ttd", calptr->SampleLinearity[aramp]);
2630             lm_message("ttd", calptr->SampleLinearity[aramp]);
2631             lm_message("ttd", calptr->SampleLinearity[aramp]);
2632             lm_message("ttd", calptr->SampleLinearity[aramp]);
2633             lm_message("ttd", calptr->SampleLinearity[aramp]);
2634             lm_message("ttd", calptr->SampleLinearity[aramp]);
2635             lm_message("ttd", calptr->SampleLinearity[aramp]);
2636             lm_message("ttd", calptr->SampleLinearity[aramp]);
2637             lm_message("ttd", calptr->SampleLinearity[aramp]);
2638             lm_message("ttd", calptr->SampleLinearity[aramp]);
2639             lm_message("ttd", calptr->SampleLinearity[aramp]);
2640             lm_message("ttd", calptr->SampleLinearity[aramp]);

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_cal.c

DATE

5/23/89

PAGE #

TIME

4:41:30 pm

23/170

```

LINE # SOURCE TEXT
2641
2642   lm_message("tstd Edge Offset", leader, lineno++);
2643   for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2644       lm_message("  td", calptr->EdgeOffset[aramp][edge]);
2645   lm_message("\n");
2646
2647   lm_message("tstd Edge Slope", leader, lineno++);
2648   for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2649       lm_message("  td", calptr->EdgeSlope[aramp][edge]);
2650   lm_message("\n");
2651
2652   lm_message("tstd Edge Linearity", leader, lineno++);
2653   for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2654       lm_message("  td", calptr->EdgeLinearity[aramp][edge]);
2655   lm_message("\n");
2656
2657   lm_message("tstd Edge 7 Minimum Threshold td\n", leader, lineno++);
2658   calptr->Edge7MinThresh[aramp];
2659
2660   lm_message("tstd Edge 7 Slope td\n", leader, lineno++);
2661   calptr->Edge7Slope[aramp];
2662
2663   lm_message("tstd Edge 7 Linearity td\n", leader, lineno++);
2664   calptr->Edge7Linearity[aramp];
2665   }
2666   for(aramp = 0; aramp < NUMBER_OF_ARAMPS; aramp++)
2667   {
2668       lm_message("tstd Sample Ramp td\n", leader, lineno++, aramp);
2669
2670       lm_message("tstd Sample Minimum Threshold td\n", leader, lineno++,
2671               calptr->SampleMinThresh[aramp]);
2672
2673       lm_message("tstd Sample Minimum Delay td\n", leader, lineno++,
2674               calptr->SampleMinDelay[aramp]);
2675
2676       lm_message("tstd Sample Maximum Delay td\n", leader, lineno++,
2677               calptr->SampleMaxDelay[aramp]);
2678
2679       lm_message("tstd Sample Slope td\n", leader, lineno++,
2680               calptr->SampleSlope[aramp]);
2681
2682       lm_message("tstd Sample Linearity td\n", leader, lineno++,
2683               calptr->SampleLinearity[aramp]);
2684
2685       lm_message("tstd Sample Offset", leader, lineno++);
2686       for(aramp = 0; aramp < NUMBER_OF_ARAMPS; aramp++)
2687           lm_message("  td", calptr->SampleOffset[aramp][aramp]);
2688       lm_message("\n");
2689
2690       lm_message("tstd Early Sample Minimum Delay td\n", leader, lineno++,
2691               calptr->EarlySampleMinDelay[aramp]);
2692
2693       lm_message("tstd Early Sample Maximum Delay td\n", leader, lineno++,
2694               calptr->EarlySampleMaxDelay[aramp]);
2695
2696       lm_message("tstd Early Sample Offset td\n", leader, lineno++,
2697               calptr->EarlySampleOffset[aramp]);
2698   }
2699
2700   lm_message("tstd Delay Line Setting td\n", leader, lineno++,
2701           calptr->DelayDelay);
2702
2703   lm_message("tstd END\n", leader, lineno++);
2704
2705
2706
2707   /*
2708   *   tmg_process_and_transfer_calib_structure(temptr, realptr)
2709   *   This routine processes all the data in the temporary calibration
2710   *   structure and transfers it to the real one.
2711   *   Inputs: pointers to the temporary and real calib structures
2712   *   Outputs: returns SUCCESS or FAILURE
2713   */
2714   tmg_process_and_transfer_calib_structure(temptr, realptr)
2715   struct CALIB *temptr, *realptr;
2716   {
2717       int i;
2718       char *from, *to;
2719       tmg_complete_calib_structure(temptr);
2720       from = (char *)temptr;
2721       to = (char *)realptr;
2722       for(i = 0; i < sizeof(struct CALIB); i++)
2723           *to++ = *from++;
2724       return SUCCESS;
2725   }
2726
2727
2728
2729   tmg_is_calibrated()
2730   {
2731       return (calib.CalCompleted == CALIBRATION_DONE);
2732   }
2733
2734   static long tmg_save_test_mode, tmg_save_delay_reg;
2735
2736   tmg_restore_calibration()
2737   {
2738       if (!tmg_is_calibrated()) {
2739           if (tmg_fast_calibrate() != SUCCESS)
2740               return FAILURE;
2741       } else {
2742           tmg_set_test_mode(tmg_save_test_mode);
2743           tmg_set_delay_reg(tmg_save_delay_reg);
2744       }
2745       return SUCCESS;
2746   }
2747
2748   tmg_save_calibration()
2749   {
2750       tmg_save_test_mode = tmg_get_test_mode();
2751       tmg_save_delay_reg = tmg_get_delay_reg();
2752   }
2753
2754
2755   /*
2756   *   tmg_set_timing(period, edgetime, settime, settime, spw)
2757   *   Inputs:   period      clock period in picoseconds
2758   *            edgetime     array of 7 edge times in picoseconds
2759   *            settime      sample start time in picoseconds
2760   *            settime      latest expected sample start time in picoseconds

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_cal.c

DATE

5/23/89

PAGE #

TIME

4:41:30 pm

24/171

```

LINE # SOURCE TEXT
2761 ** (used to select sample ramp)
2762 ** sample pulse width in picoseconds
2763 **
2764
2765 tmg_set_timing(period, edgetime, astime, settime, spw)
2766 long period, edgetime, astime, settime, spw;
2767 {
2768     static char me[] = "tmg_set_timing";
2769     long startdeadtme,
2770     long enddeadtme,
2771     long thresholds[NUMBER_OF_EDGES + 1],
2772     long ramp, edge, thr, slope, offset, arange, sthreshold,
2773     long edge_limit,
2774     long sample_delta, arampmindelay, minst,
2775
2776     /* add start deadtime to andy's numbers */
2777
2778     /* select edge ramp */
2779
2780     for (ramp = 0; ramp < NUMBER_OF_RAMPS; ++ramp) {
2781         long_yet_ramp_dead_time(period, ramp, &startdeadtme, &enddeadtme);
2782         slope = calib.Edgeslope[ramp][0];
2783         offset = calib.EdgeOffset[ramp][0];
2784         thr = tmg_predict_threshold(period - enddeadtme, slope, offset);
2785         if ((thr >= calib.EdgeMinThresh[ramp][0]) && (thr < 255))
2786             break;
2787     }
2788
2789     if (ramp >= NUMBER_OF_RAMPS) {
2790         ln_error("%s: Requested period out of range\n", me);
2791         return FAILURE;
2792     }
2793
2794     /* Now we have selected our edge ramp, calculate thresholds */
2795
2796     /* Now, all edge settings are relative to time 0 (at minst) */
2797     for (edge = 0; edge < NUMBER_OF_EDGES; edge++) {
2798         edge_limit = tmg_predict_period(thr,
2799             calib.Edgeslope[ramp][edge], calib.EdgeOffset[ramp][edge]),
2800         thresholds[edge] = tmg_predict_threshold(edgetime[edge],
2801             calib.Edgeslope[ramp][edge], 0) + calib.EdgeMinThresh[ramp][edge];
2802         if (edgetime[edge] > edge_limit) {
2803             if (tmg_cal_debug & 0x4000000) {
2804                 ln_warning("%s: Edge %d (slope=%d thr %d) too large for ramp %d\n",
2805                     me, edge, edgetime[edge], edge_limit, thresholds[edge], ramp);
2806                 thresholds[edge] = 255; /* shut the sucker off! */
2807             }
2808         }
2809     }
2810
2811     /* select sample mode */
2812     tmgptr->sample_mode = EDGE7RAMPLETRIGGERMODE;
2813
2814     /* select sample range */
2815     sample_delta = settime - astime;
2816     for (arange = 0; arange < NUMBER_OF_RAMPS; ++arange) {
2817         if (sample_delta <
2818             (255 - END_DEAD_TIME_FUDGE - calib.SampleMinThresh[arange])
2819             * calib.SampleSlope[arange])
2820             break;
2821     }
2822     if (arange >= NUMBER_OF_RAMPS) {
2823         ln_error("%s: sample time too big\n", me);
2824         return FAILURE;
2825     }
2826     sthreshold = calib.SampleMinThresh[arange];
2827
2828     /* now calculate the minimum legal sample time */
2829     arampmindelay = tmg_predict_period(calib.SampleMinThresh[arange],
2830         calib.SampleSlope[arange], calib.SampleOffset[ramp][arange]);
2831     minst = arampmindelay
2832         + tmg_predict_period(calib.Edge7MinThresh[ramp],
2833             calib.Edge7Slope[ramp], 0);
2834
2835     /* select edge 7 threshold */
2836     thresholds[6] = tmg_predict_threshold(astime + minst,
2837         calib.Edge7Slope[ramp], arampmindelay);
2838
2839     /* set clock period */
2840     if (tmg_set_period(ramp, &period) != SUCCESS) {
2841         tmg_report_failure(me, "tmg_set_period");
2842         return FAILURE;
2843     }
2844
2845     /* set edges and edge ramp */
2846     if (ln_tmg_set_ramp_edge_settings(ramp, thresholds) != SUCCESS) {
2847         tmg_report_failure(me, "ln_tmg_set_ramp_edge_settings");
2848         return FAILURE;
2849     }
2850
2851     /* set sample range and threshold */
2852     if (ln_tmg_set_rising_sample(arange, sthreshold) != SUCCESS) {
2853         tmg_report_failure(me, "ln_tmg_set_rising_sample");
2854         return FAILURE;
2855     }
2856
2857     /* set sample pulse width */
2858     if (tmg_set_sample_width(period, slope, spw) != SUCCESS) {
2859         tmg_report_failure(me, "tmg_set_sample_width");
2860         return FAILURE;
2861     }
2862
2863     if (tmg_cal_debug & 0x4000000) {
2864         ln_message("%s:\n", me);
2865         ln_message("\t period = %d\n", period);
2866         ln_message("\t ramp = %d\n", ramp);
2867         for (edge = 0; edge <= NUMBER_OF_EDGES; edge++) {
2868             ln_message("\t Edge %d threshold = %d\n", edge, thresholds[edge]);
2869         }
2870         ln_message("\t sample range = %d\n", arange);
2871         ln_message("\t sample threshold = %d\n", sthreshold);
2872         ln_message("\t sample pulse width register = %d\n",
2873             tmg_get_sample_width_reg(sample_width));
2874     }
2875
2876     return SUCCESS;
2877 }
2878
2879 /* tmg_fast_calibrate() uses conservative default values for slopes */
2880

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_cal.c | DATE 5/23/89 | PAGE # 25/172 |
|--|------|--|-----------------|------------------|
| LINE # | | SOURCE TEXT | | |
| 2881 | 2882 | /* etc, then really measures and sets the proper delay. */ | | |
| 2883 | 2884 | tmg_fast_calibrate() | | |
| 2885 | 2886 | { | | |
| 2887 | 2888 | static char me[] = "tmg_fast_calibrate"; | | |
| 2889 | 2890 | a... struct CALIB default_calib; | | |
| 2891 | 2892 | static long fast_cal_delay = 0; | | |
| 2893 | 2894 | long errors; | | |
| 2895 | 2896 | register long edge; | | |
| 2897 | 2898 | bcopy(&default_calib, &calib, sizeof(calib)); | | |
| 2899 | 2900 | tmg_set_slot_count(2); | | |
| 2901 | 2902 | tmg_set_test_mode(1); | | |
| 2903 | 2904 | { | | |
| 2905 | 2906 | if (!fast_cal_delay) { | | |
| 2907 | 2908 | tmg_set_delay(14); | | |
| 2909 | 2910 | /* delay = 5ns + 14ns */ | | |
| 2911 | 2912 | if (tmg_measure_ramp(01, 01, &calib, &errors) != SUCCESS) { | | |
| 2913 | 2914 | tmg_report_failure(me, "tmg_measure_ramp(0,0)"); | | |
| 2915 | 2916 | tmg_set_slot_count(1); | | |
| 2917 | 2918 | tmg_set_test_mode(0); | | |
| 2919 | 2920 | return FAILURE; | | |
| 2921 | 2922 | } | | |
| 2923 | 2924 | for (edge = 1; edge < NUMBER_OF_EDGES; ++edge) { | | |
| 2925 | 2926 | calib.EdgeSlope[0][edge] = calib.EdgeSlope[0][0]; | | |
| 2927 | 2928 | calib.EdgeOffset[0][edge] = calib.EdgeOffset[0][0]; | | |
| 2929 | 2930 | } | | |
| 2931 | 2932 | if (tmg_calibrate_delay(&calib) != SUCCESS) { | | |
| 2933 | 2934 | tmg_report_failure(me, "tmg_calibrate_delay"); | | |
| 2935 | 2936 | tmg_set_slot_count(1); | | |
| 2937 | 2938 | tmg_set_test_mode(0); | | |
| 2939 | 2940 | return FAILURE; | | |
| 2941 | 2942 | } | | |
| 2943 | 2944 | fast_cal_delay = tmg_get_delay(); | | |
| 2945 | 2946 | } else { | | |
| 2947 | 2948 | tmg_set_delay(fast_cal_delay); | | |
| 2949 | 2950 | } | | |
| 2951 | 2952 | tmg_set_slot_count(1); | | |
| 2953 | 2954 | tmg_set_test_mode(0); | | |
| 2955 | 2956 | return SUCCESS; | | |
| 2957 | 2958 | } | | |
| 2959 | 2960 | /* The following functions were put here for Robert's benefit */ | | |
| 2961 | 2962 | int tmg_effclear(void) | | |
| 2963 | 2964 | { | | |
| 2965 | 2966 | /* Clears edge and sample flip flops | | |
| 2967 | 2968 | Inputs: none | | |
| 2969 | 2970 | Returns: SUCCESS or FAILURE | | |
| 2971 | 2972 | /* | | |
| 2973 | 2974 | int tmg_effclear() /* clear the cal flip flops */ | | |
| 2975 | 2976 | { | | |
| 2977 | 2978 | int returncode; | | |
| 2979 | 2980 | tmgptr->as_if_clearl = 0; | | |
| 2981 | 2982 | if(tmgptr->edge_cal & calif) /* see if any still set */ | | |
| 2983 | 2984 | { | | |
| 2985 | 2986 | lm_error("Could not clear cal flip flops\n"); | | |
| 2987 | 2988 | returncode = FAILURE; | | |
| 2989 | 2990 | } | | |
| 2991 | 2992 | else | | |
| 2993 | 2994 | returncode = SUCCESS; | | |
| 2995 | 2996 | tmgptr->as_if_clearl = 1; | | |
| 2997 | 2998 | return(returncode); | | |
| 2999 | 3000 | } | | |
| 3001 | 3002 | /* | | |
| 3003 | 3004 | int tmg_play(timeout) | | |
| 3005 | 3006 | { | | |
| 3007 | 3008 | /* Initiates pattern presentation | | |
| 3009 | 3010 | Inputs: none | | |
| 3011 | 3012 | Returns: SUCCESS or FAILURE | | |
| 3013 | 3014 | /* | | |
| 3015 | 3016 | int tmg_play(timeout) /* start a presentation */ | | |
| 3017 | 3018 | { | | |
| 3019 | 3020 | long timeout; | | |
| 3021 | 3022 | { | | |
| 3023 | 3024 | if(tmg_initiate_play() != SUCCESS) | | |
| 3025 | 3026 | return(FAILURE); | | |
| 3027 | 3028 | if(tmg_complete_play(timeout) != SUCCESS) | | |
| 3029 | 3030 | return(FAILURE); | | |
| 3031 | 3032 | else | | |
| 3033 | 3034 | return(SUCCESS); | | |
| 3035 | 3036 | } | | |
| 3037 | 3038 | } | | |
| 3039 | 3040 | /* | | |
| 3041 | 3042 | int tmg_initiate_play(void) | | |
| 3043 | 3044 | { | | |
| 3045 | 3046 | /* This routine turns clocks on and starts a presentation. | | |
| 3047 | 3048 | Inputs: none | | |
| 3049 | 3050 | Outputs: function returns SUCCESS or FAILURE | | |
| 3051 | 3052 | /* | | |
| 3053 | 3054 | int tmg_initiate_play() | | |
| 3055 | 3056 | { | | |
| 3057 | 3058 | if(tmg_clockoff() != SUCCESS) | | |
| 3059 | 3060 | { | | |
| 3061 | 3062 | lm_error("tmg_initiate_play: tmg_clockoff returned error\n"); | | |
| 3063 | 3064 | return(FAILURE); | | |
| 3065 | 3066 | } | | |
| 3067 | 3068 | if(tmg_clockon() != SUCCESS) | | |
| 3069 | 3070 | { | | |
| 3071 | 3072 | lm_error("play: tmg_clockon returned error\n"); | | |
| 3073 | 3074 | return(FAILURE); | | |
| 3075 | 3076 | } | | |
| 3077 | 3078 | tmgptr->pattern_intr_enable = 0; /* just in case */ | | |
| 3079 | 3080 | tmgptr->pattern_intr_enable = 1; /* enable EOP interrupt */ | | |
| 3081 | 3082 | tmgptr->start_pattern_play = 1; /* start presentation */ | | |
| 3083 | 3084 | return(SUCCESS); | | |
| 3085 | 3086 | } | | |
| 3087 | 3088 | } | | |
| 3089 | 3090 | /* | | |
| 3091 | 3092 | int tmg_complete_play(long timeout) | | |
| 3093 | 3094 | { | | |
| 3095 | 3096 | /* This function waits for a presentation to complete by | | |
| 3097 | 3098 | waiting for the EOP interrupt from the Timing Generator. | | |
| 3099 | 3100 | { | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_cal.c

DATE 5/23/89

PAGE #

TIME 4:41:30 pm

26/173

```

LINE # SOURCE TEXT
3001 * Inputs: timeout is ms
3002 * Outputs: function returns SUCCESS or FAILURE
3003 */
3004 int tmg_complete_play(timeout)
3005 {
3006     long start;
3007     int returncode = SUCCESS;
3008     start = lm_time();
3009     #ifndef DIAGS
3010     while (!cpu_tmg_interrupt_status())
3011     #else
3012     while (!TMC_INT) /* wait for interrupt */
3013     #endif
3014     {
3015         if((lm_time() - start) > timeout)
3016         {
3017             lm_error("tmg_complete_play: timeout waiting for access mode\n");
3018             returncode = FAILURE;
3019             break;
3020         }
3021         tmgptr->pattern_intr_enable = 0; /* clear EOP interrupt */
3022         if(tmg_clockoff() != SUCCESS)
3023             returncode = FAILURE;
3024         return(returncode);
3025     }
3026 }
3027
3028 /* int tmg_clockoff(void)
3029 * shuts clock off
3030 * Input: nothing
3031 * Output: SUCCESS or FAILURE
3032 */
3033 int tmg_clockoff()
3034 {
3035     long start;
3036     tmgptr->clock_enable = 0;
3037     start = lm_time();
3038     while(tmgptr->clock_on)
3039     {
3040         if((lm_time() - start) > 10)
3041         {
3042             if(tmgptr->clock_select > 2) /* must be external clock */
3043                 tmgptr->clock_sync_clear1 = 0;
3044             break;
3045         }
3046     }
3047     if(tmgptr->clock_on)
3048         return(FAILURE);
3049     tmgptr->clock_sync_clear1 = 0;
3050     return(SUCCESS);
3051 }
3052
3053 /* int tmg_clockon(void)
3054 * turns clock on
3055 * Input: nothing
3056 * Output: SUCCESS or FAILURE
3057 */
3058 int tmg_clockon()
3059 {
3060     long start;
3061     int playmode;
3062     playmode = tmgptr->backplane_mode;
3063     tmgptr->clock_enable = 0;
3064     tmgptr->clock_sync_clear1 = 1;
3065     tmgptr->clock_enable = 1;
3066     /* There is one weird condition that if the state machine is
3067     /* caught in play mode and the DM is reset, when the clock
3068     /* turns on, PLAY goes low and turns it off again.
3069     if(playmode)
3070     {
3071         start = lm_time();
3072         while(!tmgptr->clock_on)
3073             if((lm_time() - start) > 10)
3074             {
3075                 tmgptr->clock_enable = 0;
3076                 tmgptr->clock_enable = 1;
3077                 break;
3078             }
3079     }
3080     start = lm_time();
3081     while(!tmgptr->clock_on)
3082     {
3083         if((lm_time() - start) > 10)
3084         {
3085             lm_error("tmg_clockon: did not turn on\n");
3086             return(FAILURE);
3087         }
3088     }
3089     return(SUCCESS);
3090 }
3091
3092 tmg_set_defaults()
3093 {
3094     *REG(0) = 0x10; /* not OBSERVE, not RESET_OUT */
3095     *REG(1) = 0x00; /* 25 MHz
3096     *REG(2) = 0x2c; /* Select divide by 2
3097     *REG(3) = 0x0ff; /* Select fast ramps, no lanes
3098     *REG(4) = 0x00; /* Test mode 1, 18ns delay
3099     *REG(5) = 0x52; /* 10ns dump, ext clock
3100     *REG(6) = 0x07; /* 15 clock sample width
3101     *REG(7) = 0x07; /* minimum threshold EDGE[0]
3102     *REG(16) = 0x07; /* EDGE[1]
3103     *REG(17) = 0x07; /* EDGE[2]
3104     *REG(18) = 0x07; /* EDGE[3]
3105     *REG(19) = 0x07; /* EDGE[4]
3106     *REG(20) = 0x07; /* EDGE[5]
3107     *REG(21) = 0x07; /* SAMPLE Trigger
3108     *REG(22) = 0x07; /* SAMPLE
3109     *REG(23) = 0x07;
3110 }
3111
3112 int tmg_reset(do_bp_reset)
3113 {
3114     /* This function performs a power up initialization sequence.
3115     /* Inputs: do_bp_reset = TRUE if backplane reset desired, else FALSE
3116     /* Outputs: function returns SUCCESS or FAILURE
3117 }
3118
3119
3120

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_cal.c

DATE 5/23/89
TIME 4:41:30 pm

PAGE #
27/174

```

LINE # SOURCE TEXT
3121 int
3122 tmg_reset(do_bp_reset)
3123 int do_bp_reset;
3124 {
3125     int returncode = SUCCESS;
3126
3127     #ifdef DIAGS
3128     if(!Host)
3129         return(returncode);
3130
3131     if (do_bp_reset) pac_info_init();
3132     #endif DIAGS
3133     tmg_set_defaults();
3134     tmgptr->tmg_resetl = 0;
3135     if (do_bp_reset == TRUE)
3136         tmgptr->backplane_resetl = 0;
3137
3138     lm_delay(1); /* wait for PLL to lock */
3139
3140     /* some synchronous magic here... */
3141     if(tmg_clockoff() != SUCCESS)
3142     {
3143         lm_error("Reset: cannot turn clock off.\n");
3144         returncode = FAILURE;
3145     }
3146     if(tmg_clockon() != SUCCESS)
3147     {
3148         lm_error("Reset-1: cannot turn clock on.\n");
3149         returncode = FAILURE;
3150     }
3151     if(tmg_clockoff() != SUCCESS)
3152     {
3153         lm_error("Reset: cannot turn clock off.\n");
3154         returncode = FAILURE;
3155     }
3156
3157     tmgptr->tmg_resetl = 1; /* remove TMG reset */
3158     if(tmg_play((long)TIMEOUT) != SUCCESS)
3159     {
3160         lm_error("Reset: cannot play.\n");
3161         returncode = FAILURE;
3162     }
3163
3164     tmg_set_test_mode(0);
3165
3166     if (do_bp_reset == TRUE)
3167     {
3168         if(tmg_clockon() != SUCCESS)
3169         {
3170             lm_error("Reset-2: cannot turn clock on.\n");
3171             returncode = FAILURE;
3172         }
3173         lm_delay(5);
3174         if(tmg_clockoff() != SUCCESS)
3175         {
3176             lm_error("Reset: cannot turn clock off.\n");
3177             returncode = FAILURE;
3178         }
3179         lm_delay(5);
3180         tmgptr->backplane_resetl = 1; /* remove backplane reset */
3181
3182         /* This next section of code checks for backplane errors
3183          * and attempts to remove them by probing the PACs in every
3184          * lane (whether they are there or not). This is because it
3185          * is possible for the TMC to have a bogus refresh error
3186          * after the reset is de-asserted. Probing the error register
3187          * will remove this error. If errors still exist after the
3188          * probing takes place, the function returns FAILURE.
3189          */
3190         if(tmgptr->lane_intr != 0)
3191         {
3192             (void)lm_write_probe(0x0c1002801, 01);
3193             (void)lm_write_probe(0x0c1002801, 01);
3194             (void)lm_write_probe(0x0c1002801, 01);
3195             (void)lm_write_probe(0x0c1002801, 01);
3196         }
3197     }
3198
3199     if(bp_mode() == PLAY_MODE)
3200     {
3201         (void)lm_error("Backplane is still in play mode after TMC init.\n");
3202         return FAILURE;
3203     }
3204
3205     #ifdef DIAGS
3206     /* Check for backplane errors */
3207     if(diag_clear_errors() != SUCCESS)
3208     {
3209         return FAILURE;
3210     }
3211     #endif
3212     return returncode;
3213 }
3214
3215 diag_tmg_reset(do_bp_reset)
3216 int do_bp_reset;
3217 {
3218     int returncode;
3219     static long edgetime[] = {
3220         5000, /* edge 0 */
3221         5000, /* edge 1 */
3222         5000, /* edge 2 */
3223         15000, /* edge 3 */
3224         5000, /* edge 4 */
3225         0, /* edge 5 */
3226     };
3227
3228     returncode = tmg_reset(do_bp_reset);
3229
3230     if (tmg_restore_calibration() != SUCCESS) {
3231         (void)lm_error("Unable to restore calibration\n");
3232         return FAILURE;
3233     }
3234     tmg_set_slot_count(1);
3235     if (tmg_set_timing(1000001, edgetime, 01, 1000001, 25000001) != SUCCESS) {
3236         lm_error("Could not set edges\n");
3237         returncode = FAILURE;
3238     }
3239 }
3240

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_cal.c | DATE 5/23/89 | PAGE # 28/175 |
|--|--|---|-----------------|------------------|
| LINE # | | SOURCE TEXT | | |
| 3241 | | return(returacode); | | |
| 3242 | | } | | |
| 3243 | | tmg_report_failure(called_from, failing_func) | | |
| 3244 | | char *called_from, *failing_func; | | |
| 3245 | | { | | |
| 3246 | | return lm_error("ts: ts failed\n", called_from, failing_func); | | |
| 3247 | | } | | |
| 3248 | | } | | |
| 3249 | | /* | | |
| 3250 | | void tmg_getc(ts, sprompt) | | |
| 3251 | | { | | |
| 3252 | | /* | | |
| 3253 | | INPUT: ts = input character | | |
| 3254 | | sprompt = address of input prompt | | |
| 3255 | | OUTPUT: none | | |
| 3256 | | DESCRIPTION: gets char value from keyboard. | | |
| 3257 | | */ | | |
| 3258 | | void | | |
| 3259 | | tmg_getc(c, prompt) | | |
| 3260 | | char *c, | | |
| 3261 | | char *prompt, | | |
| 3262 | | { | | |
| 3263 | | char reply[2], | | |
| 3264 | | char buffer[1024], | | |
| 3265 | | /* Get input from keyboard */ | | |
| 3266 | | sprintf(buffer, "Enter ts : ", prompt); | | |
| 3267 | | #ifdef DIAGS | | |
| 3268 | | lm_get_input(buffer, reply, 2); | | |
| 3269 | | #endif DIAGS | | |
| 3270 | | *c = reply[0]; | | |
| 3271 | | } | | |
| 3272 | | } | | |
| 3273 | | /* | | |
| 3274 | | Debug Flag routines*/ | | |
| 3275 | | /* | | |
| 3276 | | Allow the user in "wizard" mode to set/reset the various | | |
| 3277 | | debug flags (tmg_cal_debug) to turn on/off all the wonderful | | |
| 3278 | | messages available. This is mostly a software debugging aid, but | | |
| 3279 | | can come in handy debugging hardware, too. | | |
| 3280 | | */ | | |
| 3281 | | #define WHAT(x) (tmg_cal_debug & (x) ? "ON" : "OFF") | | |
| 3282 | | #define FLIP(x) (tmg_cal_debug ^= (x)) | | |
| 3283 | | long tmg_modify_edge_related_debug_flag() | | |
| 3284 | | { | | |
| 3285 | | char answer; | | |
| 3286 | | do | | |
| 3287 | | { | | |
| 3288 | | answer = 0; | | |
| 3289 | | lm_message("\nEDGE related debug flags are currently set as follows:\n"); | | |
| 3290 | | lm_message("\tSingle stop (ts)\n", WHAT(1)); | | |
| 3291 | | lm_message("\tPrint Arguments (ts)\n", WHAT(2)); | | |
| 3292 | | lm_message("\tPrint Process Messages (ts)\n", WHAT(4)); | | |
| 3293 | | lm_message("\tPrint Output Messages (ts)\n", WHAT(8)); | | |
| 3294 | | lm_message("\tPrint Starting Points (ts)\n", WHAT(0x10)); | | |
| 3295 | | tmg_getc(&answer, "flag to toggle, 'Q' to end"); | | |
| 3296 | | switch(answer) | | |
| 3297 | | { | | |
| 3298 | | case 's': | | |
| 3299 | | case 'S': FLIP(1); break; | | |
| 3300 | | case 'a': | | |
| 3301 | | case 'A': FLIP(2); break; | | |
| 3302 | | case 'p': | | |
| 3303 | | case 'P': FLIP(4); break; | | |
| 3304 | | case 'o': | | |
| 3305 | | case 'O': FLIP(8); break; | | |
| 3306 | | case 't': | | |
| 3307 | | case 'T': FLIP(0x10); break; | | |
| 3308 | | default: break; | | |
| 3309 | | } | | |
| 3310 | | } while((answer != 'q') && (answer != 'Q')); | | |
| 3311 | | return SUCCESS; | | |
| 3312 | | } | | |
| 3313 | | long tmg_modify_sample_related_debug_flag() | | |
| 3314 | | { | | |
| 3315 | | char answer; | | |
| 3316 | | do | | |
| 3317 | | { | | |
| 3318 | | answer = 0; | | |
| 3319 | | lm_message("\nSAMPLE related debug flags are currently set as follows:\n"); | | |
| 3320 | | lm_message("\tSingle stop (ts)\n", WHAT(0x100)); | | |
| 3321 | | lm_message("\tPrint Arguments (ts)\n", WHAT(0x200)); | | |
| 3322 | | lm_message("\tPrint Process Messages (ts)\n", WHAT(0x400)); | | |
| 3323 | | lm_message("\tPrint Output Messages (ts)\n", WHAT(0x800)); | | |
| 3324 | | lm_message("\tPrint Starting Points (ts)\n", WHAT(0x1000)); | | |
| 3325 | | tmg_getc(&answer, "flag to toggle, 'Q' to end"); | | |
| 3326 | | switch(answer) | | |
| 3327 | | { | | |
| 3328 | | case 's': | | |
| 3329 | | case 'S': FLIP(0x100); break; | | |
| 3330 | | case 'a': | | |
| 3331 | | case 'A': FLIP(0x200); break; | | |
| 3332 | | case 'p': | | |
| 3333 | | case 'P': FLIP(0x400); break; | | |
| 3334 | | case 'o': | | |
| 3335 | | case 'O': FLIP(0x800); break; | | |
| 3336 | | case 't': | | |
| 3337 | | case 'T': FLIP(0x1000); break; | | |
| 3338 | | default: break; | | |
| 3339 | | } | | |
| 3340 | | } while((answer != 'q') && (answer != 'Q')); | | |
| 3341 | | return SUCCESS; | | |
| 3342 | | } | | |
| 3343 | | long tmg_modify_delayline_related_debug_flag() | | |
| 3344 | | { | | |
| 3345 | | char answer; | | |
| 3346 | | do | | |
| 3347 | | { | | |
| 3348 | | answer = 0; | | |
| 3349 | | lm_message("\nDELAY LINE related debug flags are currently set as follows:\n"); | | |
| 3350 | | lm_message("\tCalibrate delay output Messages (ts)\n", WHAT(0x10000)); | | |
| 3351 | | lm_message("\tCompute delay process Messages (ts)\n", WHAT(0x20000)); | | |
| 3352 | | tmg_getc(&answer, "flag to toggle, 'Q' to end"); | | |
| 3353 | | switch(answer) | | |
| 3354 | | { | | |
| 3355 | | case 'c': | | |
| 3356 | | case 'C': FLIP(0x10000); break; | | |
| 3357 | | case 'p': | | |
| 3358 | | case 'P': FLIP(0x20000); break; | | |
| 3359 | | default: break; | | |
| 3360 | | } | | |
| 3361 | | } while((answer != 'q') && (answer != 'Q')); | | |
| 3362 | | return SUCCESS; | | |
| 3363 | | } | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_cal.c

DATE 5/23/89
TIME 4:41:30 pm
PAGE # 29/176

```

LINE # SOURCE TEXT
3361 ln_message("\tadjust delay process messages (%s)\n", WEAT(0x40000));
3362 tmg_getc(&answer, "flag to toggle, 'Q' to end");
3363 switch(answer)
3364 {
3365     case 'C':
3366         case 'C': FLIP(0x10000); break;
3367     case 'O':
3368         case 'O': FLIP(0x20000); break;
3369     case 'A':
3370         case 'A': FLIP(0x40000); break;
3371     default: break;
3372 }
3373 } while((answer != 'q') && (answer != 'Q'));
3374 return SUCCESS;
3375 }
3376 }
3377 long tmg_modify_misc_related_debug_flag()
3378 {
3379     char answer;
3380     do
3381     {
3382         answer = 0;
3383         ln_message("\tMISCELLANEOUS related debug flags are currently set as follows:\n");
3384         ln_message("\tprint Calibration structure (%s)\n", WEAT(0x800000));
3385         ln_message("\tprint line fit arguments (%s)\n", WEAT(0x1000000));
3386         ln_message("\tprint line fit outputs (%s)\n", WEAT(0x2000000));
3387         ln_message("\ttmg_set_timing() Messages (%s)\n", WEAT(0x4000000));
3388         ln_message("\ttmg_set_sample_width() Messages (%s)\n", WEAT(0x8000000));
3389         ln_message("\ttmg_complete_calib_structure Process Messages (%s)\n", WEAT(0x10000000));
3390         ln_message("\tallow extra errors before abort (%s)\n", WEAT(0x20000000));
3391         ln_message("\tsingle step major calibration sections (%s)\n", WEAT(0x40000000));
3392         ln_message("\tprint Data logging messages (%s)\n", WEAT(0x80000000));
3393         tmg_getc(&answer, "flag to toggle, 'Q' to end");
3394         switch(answer)
3395         {
3396             case 'C':
3397                 case 'C': FLIP(0x800000); break;
3398             case 'I':
3399                 case 'I': FLIP(0x1000000); break;
3400             case 'I':
3401                 case 'I': FLIP(0x2000000); break;
3402             case 'M':
3403                 case 'M': FLIP(0x4000000); break;
3404             case 'E':
3405                 case 'E': FLIP(0x8000000); break;
3406             case 'P':
3407                 case 'P': FLIP(0x10000000); break;
3408             case 'A':
3409                 case 'A': FLIP(0x20000000); break;
3410             case 'S':
3411                 case 'S': FLIP(0x40000000); break;
3412             case 'D':
3413                 case 'D': FLIP(0x80000000); break;
3414             default: break;
3415         }
3416     } while((answer != 'q') && (answer != 'Q'));
3417     return SUCCESS;
3418 }
3419 }
3420 }
3421 }
3422 }
3423 long tmg_modify_all_debug_flag()
3424 {
3425     char answer;
3426     int done = 0;
3427     do
3428     {
3429         ln_message("Set or Reset all debug flags\n");
3430         tmg_getc(&answer, "Set, Reset, or Quit");
3431         switch(answer)
3432         {
3433             case 'S':
3434                 case 'S': tmg_cal_debug |= 0x0fffff; done++; break;
3435             case 'R':
3436                 case 'R': tmg_cal_debug &= 0; done++; break;
3437             default: break;
3438         }
3439     } while(!done);
3440     return SUCCESS;
3441 }
3442 }
3443 }
3444 extern struct calib;
3445 print_calib_structure()
3446 {
3447     if(!tmg_is_calibrated())
3448     {
3449         ln_message("\t007\007Calibration has not been successfully completed\n");
3450         ln_message("Interpret the calibration data accordingly\n");
3451     }
3452     tmg_print_calib_structure(&calib);
3453 #ifdef PS_TS_DEBUG
3454     print_ps_and_ts();
3455 #endif PS_TS_DEBUG
3456     return SUCCESS;
3457 }
3458 }
3459 #ifdef PS_TS_DEBUG
3460 print_ps_and_ts()
3461 {
3462     int x, e, i;
3463     for (x = 0; x < NUMBER_OF_ERAPS; ++x) {
3464         for (e = 0; e < (NUMBER_OF_EDGES + 1); ++e) {
3465             for (i = 0; i < 7; ++i) {
3466                 ln_message("Rtd Edt l5d l3d\n", x, e, p[x][e][i], t[x][e][i]);
3467             }
3468             ln_message("\n");
3469         }
3470     }
3471 }
3472 }
3473 #endif PS_TS_DEBUG

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_diag.c

DATE 5/23/89
TIME 4:41:33 pm

PAGE #
1/177

```

1  // SCCS_ID: tmg_diag.c rev 1.2, 5/9/89 at 15:55:24
2  //
3  //.....
4  //      tmg_diag.c
5  //      Diagnostic routines for the Timing Generator
6  //.....
7  //
8  //
9  //
10 //
11 #include "common.h"
12 #include "vrtx.h"
13 #include "cpu.h"
14 #include "lm_diags.h"
15 #include "tmg.h"
16 #include "tmg_def.h"
17 #include "mod_def.h"
18 #include "tmg_exts.h"
19 #include "id.h"
20 extern int tmg_cal_debug;
21 //
22 //
23 //      long tmg_register_test(void)
24 //      Performs register test of all read/write registers
25 //
26 //      Input: none
27 //      Returns: SUCCESS if no errors, else FAILURE
28 //
29 tmg_register_test()
30 {
31     int i, returncode;
32     static unsigned char misc[] = {0x10, 2, 3, 4, 5, 6, 7, 8};
33     unsigned char tmp;
34
35     if(tmg_clockoff() != SUCCESS) /* turn clock off so as not to interfere */
36         return(FAILURE);
37
38     returncode = SUCCESS;
39
40     /* First check to see that all bits of all registers work */
41     /* Register 0 is weird in that it has OMRSET that clears bits 0 and 1 */
42     /* so it gets its own walk1 and walk0 test. */
43     /* Register 2 is special because it is the PLL rate register and is not */
44     /* allowed to contain some values. It can contain only values between */
45     /* 1 and 129. The walking 0 test has been modified accordingly. */
46     for(i = 0; i < 7; i++)
47     {
48         switch(i)
49         {
50             case 0:
51                 if(tmg_reg0walk1(REG(i)) != SUCCESS)
52                     returncode = FAILURE;
53                 if(tmg_reg0walk0(REG(i)) != SUCCESS)
54                     returncode = FAILURE;
55                 break;
56             case 2:
57                 if(tmg_walk1(REG(i)) != SUCCESS)
58                     returncode = FAILURE;
59                 if(tmg_reg2walk0(REG(i)) != SUCCESS)
60                     returncode = FAILURE;
61                 break;
62             default:
63                 if(tmg_walk1(REG(i)) != SUCCESS)
64                     returncode = FAILURE;
65                 if(tmg_walk0(REG(i)) != SUCCESS)
66                     returncode = FAILURE;
67                 break;
68         }
69     }
70
71     /* Next, see that all registers are independently addressable */
72     for(i = 0; i < 8; i++)
73     {
74         *REG(i) = misc[i];
75
76         for(i = 0; i < 8; i++)
77         {
78             if((tmp = *REG(i)) != misc[i])
79             {
80                 (void)lm_error("Register at %08lx: wrote %02x, read %02x\n",
81                               (u_long)(REG(i)), misc[i], tmp);
82                 returncode = FAILURE;
83             }
84         }
85     }
86
87     if (tmg_reset(TRUE) != SUCCESS) /* force hard reset after this test */
88     {
89         (void)lm_warning("Reset after register test failed.\n");
90     }
91
92     return(returncode);
93 }
94 //
95 //      long tmg_idprom_test(void)
96 //      Input: none
97 //      Returns: SUCCESS or FAILURE
98 //
99 tmg_idprom_test()
100 {
101     u_char tmp;
102     if((tmp = id_checksum((u_char *)((int)&tmgptr->id_prom[0] + 3)))
103        != ID_CHECKSUM_GOOD)
104     {
105         (void)lm_error("ID Prom checksum %02X (should be %02X)\n", tmp, ID_CHECKSUM_GOOD);
106         return(FAILURE);
107     }
108     return SUCCESS;
109 }
110 //
111 //      long tmg_ctc_test(void)
112 //      Input: none
113 //      Returns: SUCCESS or FAILURE
114 //
115 //      Note: this routine relies on the clocks working.
116 //      It will be hard to know if the CTC counts wrong or
117 //      if the PLL generates the wrong clock frequency.
118 //
119 tmg_ctc_test()
120 {

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_diag.c | DATE 5/23/89 | PAGE # 2/178 |
|--|---|------------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 121 | int returncode, i; | | | |
| 122 | int ctrl[3], ctr2[3]; | | | |
| 123 | unsigned long start; | | | |
| 124 | int zero = 0; | | | |
| 125 | | | | |
| 126 | returncode = SUCCESS; | | | |
| 127 | if(tmg_clockoff() != SUCCESS) /* shut clock off if on */ | | | |
| 128 | { | | | |
| 129 | (void)lm_error("Cannot continue CTC Test with broken clock.\n"); | | | |
| 130 | return(FAILURE); | | | |
| 131 | } | | | |
| 132 | | | | |
| 133 | tmgptr->pll_rate = (256 - 213) * 2; /* freq=30Mhz*213/256 */ | | | |
| 134 | tmgptr->pll_divisor = 255; /* just divide by 2 */ | | | |
| 135 | tmgptr->clock_select = 0; /* select divide by 2 */ | | | |
| 136 | | | | |
| 137 | lm_delay(2); /* lots of lock time */ | | | |
| 138 | if(tmg_clockon() != SUCCESS) /* turn it on */ | | | |
| 139 | { | | | |
| 140 | (void)lm_error("Cannot continue CTC Test, clock does not turn on.\n"); | | | |
| 141 | return(FAILURE); | | | |
| 142 | } | | | |
| 143 | tmgptr->ctc_intr_clear1 = 0; /* make sure output is false */ | | | |
| 144 | if(tmgptr->ctc_intr) | | | |
| 145 | { | | | |
| 146 | (void)lm_error("CTC Test: Interrupt flip flop did not clear.\n"); | | | |
| 147 | } | | | |
| 148 | tmgptr->ctc_register[3].value = CTRL0CH; /* control word */ | | | |
| 149 | tmgptr->ctc_register[0].value = 1; /* very short count, lab */ | | | |
| 150 | tmgptr->ctc_register[0].value = zero; /* mab */ | | | |
| 151 | | | | |
| 152 | tmgptr->ctc_intr_clear1 = 1; /* trigger timer */ | | | |
| 153 | for(i = 0; i < 10; i++) /* wait just a little while */ | | | |
| 154 | | | | |
| 155 | if(!tmgptr->ctc_intr) | | | |
| 156 | { | | | |
| 157 | (void)lm_error("CTC Test: Interrupt failure after short count.\n"); | | | |
| 158 | returncode = FAILURE; | | | |
| 159 | } | | | |
| 160 | | | | |
| 161 | tmgptr->ctc_intr_clear1 = 0; /* make sure output is false */ | | | |
| 162 | if(tmgptr->ctc_intr) | | | |
| 163 | { | | | |
| 164 | (void)lm_error("CTC Test: Interrupt flip flop did not clear.\n"); | | | |
| 165 | returncode = FAILURE; | | | |
| 166 | } | | | |
| 167 | | | | |
| 168 | tmgptr->ctc_register[3].value = CTRL0CH; /* control word */ | | | |
| 169 | tmgptr->ctc_register[0].value = 0x00; /* lab of 1000 */ | | | |
| 170 | tmgptr->ctc_register[0].value = 0x03; /* mab of 1000 */ | | | |
| 171 | | | | |
| 172 | tmgptr->ctc_register[3].value = CTRL1CH; /* control word */ | | | |
| 173 | tmgptr->ctc_register[1].value = zero; /* max count 2 ** 32 */ | | | |
| 174 | tmgptr->ctc_register[1].value = zero; | | | |
| 175 | | | | |
| 176 | tmgptr->ctc_register[3].value = CTRL2CH; /* control word */ | | | |
| 177 | tmgptr->ctc_register[2].value = zero; /* max count 2 ** 32 */ | | | |
| 178 | tmgptr->ctc_register[2].value = zero; | | | |
| 179 | | | | |
| 180 | tmgptr->ctc_intr_clear1 = 1; /* start counter 0 */ | | | |
| 181 | | | | |
| 182 | start = lm_time(); | | | |
| 183 | while(!tmgptr->ctc_intr) /* wait for done */ | | | |
| 184 | if((lm_time() - start) > 10) | | | |
| 185 | { | | | |
| 186 | (void)lm_error("CTC Test: Timer test did not finish.\n"); | | | |
| 187 | returncode = FAILURE; | | | |
| 188 | break; | | | |
| 189 | } | | | |
| 190 | | | | |
| 191 | tmgptr->ctc_register[3].value = CTRL1LATCH; /* latch count/status */ | | | |
| 192 | | | | |
| 193 | /* counter 0 loaded with count of 1000 */ | | | |
| 194 | /* counters 1,2 are loaded with 0, and count down */ | | | |
| 195 | /* freq of counter 0 clock is (213/256)*30 Mhz */ | | | |
| 196 | /* freq of counters 1,2 is 30 Mhz / 4 */ | | | |
| 197 | /* should count 1000*256/4/213/8 = 2403.7 = 963h clocks */ | | | |
| 198 | /* this results in reading 10000h - 963h = f69dh from */ | | | |
| 199 | /* counter 1 and nothing from counter 2 */ | | | |
| 200 | /* Now, it is easy to get +/- one clock accuracy since the */ | | | |
| 201 | /* two clock frequencies are almost relatively prime, and */ | | | |
| 202 | /* with a little PLL error and the fact that 7 clocks is */ | | | |
| 203 | /* close to a whole clock, it is reasonable to expect that */ | | | |
| 204 | /* we could measure 963h +2,-1 clocks, or read f69dh thru */ | | | |
| 205 | /* f69eh. If we read outside this range, it is an error. */ | | | |
| 206 | | | | |
| 207 | for(i = 0; i < 3; i++) | | | |
| 208 | ctrl[i] = tmgptr->ctc_register[i].value & 0x0ff; | | | |
| 209 | if(!(ctrl[i] & 0x00)) /* null flag not set */ | | | |
| 210 | { | | | |
| 211 | if((ctrl[i] < 0x9a) (ctrl[i] > 0x9e)) /* lab first */ | | | |
| 212 | { | | | |
| 213 | (void)lm_error("CTC Test: Count test, register 1 lab = 002x, \n | | | |
| 214 | expected from 9a thru 9e.\n", ctrl[i]); | | | |
| 215 | returncode = FAILURE; | | | |
| 216 | } | | | |
| 217 | | | | |
| 218 | if(ctrl[i] != 0x0ff) /* mab next */ | | | |
| 219 | { | | | |
| 220 | (void)lm_error("CTC Test: Count test, register 1 mab = 002x, expected f6.\n", ctrl[i]); | | | |
| 221 | returncode = FAILURE; | | | |
| 222 | } | | | |
| 223 | } | | | |
| 224 | else | | | |
| 225 | { | | | |
| 226 | (void)lm_error("CTC Test: Counter 1 Null flag set.\n"); | | | |
| 227 | returncode = FAILURE; | | | |
| 228 | } | | | |
| 229 | | | | |
| 230 | for(i = 0; i < 3; i++) | | | |
| 231 | ctrl2[i] = tmgptr->ctc_register[i].value & 0x0ff; | | | |
| 232 | if(!(ctrl2[i] & 0x00)) /* null flag not set */ | | | |
| 233 | { | | | |
| 234 | (void)lm_error("CTC Test: Count test, counter 2 null flag not set.\n"); | | | |
| 235 | returncode = FAILURE; | | | |
| 236 | } | | | |
| 237 | | | | |
| 238 | return(returncode); | | | |
| 239 | | | | |
| 240 | | | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_diag.cDATE 5/23/89
TIME 4:41:33 pmPAGE #
3/179

```

LINE #
241
242 /*
243 *   int tmg_plltime(timeptr)
244 *   Measure lock time of PLL
245 *   Input: pointer to lock time
246 *   Output: time is assigned, function returns SUCCESS or FAILURE
247 *
248 *   This routine uses the S254 in a strange way
249 *   Counter 1 is used to count the time since it's clock
250 *   is FRRALL (3.75 MHz), but the gate from Counter 0 Output
251 *   must be high.
252 *   1) First a fairly slow clock is selected
253 *   2) Counter 0 is set for Mode 1 with a long count
254 *   3) As soon as it's count is started, the clock should be
255 *   turned off, leaving it's output active (gate for Counter 1)
256 *   4) Load Counter 1
257 *   5) Monkey with PLL
258 *   6) When PLL is locked, check count in Counter 1, convert
259 *   to time
260 */
261 int tmg_plltime(timeptr)
262 unsigned long *timeptr;
263 {
264     long i, uacsa, maxtries, slowuptime, slowdowntime;
265     unsigned long value, start;
266     int ctrl[3], debounced;
267     int successes, failures, successsthistime, failedthistime;
268     int upreturcode, downreturcode;
269     int zero = 0;
270     upreturcode = downreturcode = SUCCESS;
271
272     if(tmg_clockoff() != SUCCESS) /* make sure clock is off */
273     {
274         (void)lm_error("PLL Lock Time: cannot turn clock off(1)\n");
275         return(FAILURE);
276     }
277
278     tmgptr->pll_rate = (256 - 128) * 1; /* set to 30 MHz */
279     tmgptr->pll_divisor = (256 - 15) * 1; /* choose div 2 * 15 */
280     tmgptr->clock_select = 1; /* sel div 2K */
281
282     lm_delay(2); /* wait lots of time for lock */
283
284     if(!tmgptr->pll_locked)
285     {
286         (void)lm_error("PLL Lock Time: Lock indicator not functional\n");
287         return(FAILURE);
288     }
289
290     if(tmg_clockon() != SUCCESS) /* make sure clock is on */
291     {
292         (void)lm_error("PLL Lock Time: cannot turn clock on\n");
293         return(FAILURE);
294     }
295
296     tmgptr->etc_intr_clear1 = 0; /* GATE0 low */
297     tmgptr->etc_register[3].value = 0x30; /* setup mode 0 */
298     tmgptr->etc_register[0].value = zero; /* lsb for max count */
299     tmgptr->etc_register[0].value = zero; /* mab */
300
301     /* since GATE0 has no effect on OUT0, OUT0 should be low */
302     /* and furthermore, Counter0 should not be counting */
303
304     /* at this point, merely verify that Counter0 output is low */
305     maxtries = 10000;
306     while(1)
307     {
308         tmgptr->etc_register[3].value = LATCH_OUT0;
309         if((tmgptr->etc_register[0].value & 0x80))
310             break;
311         if(--maxtries == 0)
312         {
313             (void)lm_error("PLL Lock Time: Counter0 output was not low.\n");
314             return FAILURE;
315         }
316     }
317
318     /* turn the clock off to keep the PACs & PLLs happy */
319     if(tmg_clockoff() != SUCCESS) /* make sure clock is off */
320     {
321         (void)lm_error("PLL Lock Time: cannot turn clock off(2)\n");
322         return(FAILURE);
323     }
324
325     /* measure time to lock from 30 -> 60 MHz */
326     successes = failures = 0;
327     slowuptime = 0;
328     do
329     {
330         tmgptr->pll_rate = 129; /* set to 30 MHz */
331         lm_delay(2); /* wait for lock */
332         maxtries = 1000; /* cpu speed dependent */
333         successsthistime = failedthistime = FALSE;
334         tmgptr->etc_register[3].value = 0x070; /* setup mode 0 */
335         tmgptr->etc_register[1].value = zero; /* counter now running */
336         tmgptr->etc_register[1].value = zero;
337
338         CPU_DISABLE_INTERRUPTS;
339         tmgptr->pll_rate = 1; /* (256 - 256) * 1 */
340         /* slow to 60 MHz */
341
342         while(tmgptr->pll_locked == 1)
343         {
344             if(--maxtries == 0)
345             {
346                 ++failures;
347                 failedthistime = TRUE;
348                 break;
349             }
350
351             if(failures >= 100)
352             {
353                 CPU_ENABLE_INTERRUPTS;
354                 break; /* break out of loop */
355             }
356
357             if(failedthistime == TRUE)
358             {
359                 CPU_ENABLE_INTERRUPTS;
360                 continue; /* go try again */
361             }
362         }
363     }
364     else
365     {

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_diag.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:33 pm | 4/180 |

```

LINE #      SOURCE TEXT
361      ++successes; /* must have been a success */
362      successsthistime = TRUE;
363      }
364
365      debounced = 0;
366      maxtries = 1000;
367      while(!debounced) /* wait for lock and debounce, too */
368      {
369          for(i = 0; i < 50; i++) /* look for 50 samples in row */
370          {
371              if(!(--maxtries))
372              {
373                  CPU_ENABLE_INTERRUPTS;
374                  (void)lm_error("PLL Lock Time: Timeout waiting for lock true(1)\n");
375                  return FAILURE;
376              }
377              if(!tmgptr->pll_locked)
378                  break;
379              if(i == 0) /* get the time of first lock */
380              {
381                  tmgptr->ctc_register[3].value = LATCH_CNTR_1;
382                  for(i = 0; i < 3; i++)
383                      ctrl[i] = tmgptr->ctc_register[i].value & 0x0ff;
384                  if(ctrl[0] & 0x40) /* if null flag set */
385                      value = 01;
386                  else
387                  {
388                      value = ctrl[1] + (ctrl[2] << 8);
389                      if(value == 01)
390                          value = 11;
391                      else
392                          value = 0x100011 - value;
393                  }
394              }
395              if(i == 50)
396                  debounced = 1;
397          }
398          CPU_ENABLE_INTERRUPTS;
399          if(successsthistime == TRUE)
400              slowuptime += value;
401          while(successes < 10);
402      }
403      if(successes == 0)
404      {
405          upreturacode = FAILURE;
406      }
407      else
408      {
409          slowuptime = slowuptime * 2567 / (10000 * successes);
410          lm_message("slow up time = %d usens\n", slowuptime);
411      }
412
413      /* now measure time to lock from 50 -> 30 MHz */
414      successes = failures = 0;
415      slowdowntime = 0;
416      do
417      {
418          tmgptr->pll_rate = 1; /* set to 50 MHz */
419          lm_delay(2); /* wait for lock */
420          maxtries = 1000; /* cpu speed dependent */
421          successsthistime = failedthistime = FALSE;
422          tmgptr->ctc_register[3].value = 0x070; /* setup mode 0 */
423          tmgptr->ctc_register[1].value = zero; /* counter now running */
424          tmgptr->ctc_register[1].value = zero;
425
426          CPU_DISABLE_INTERRUPTS;
427          tmgptr->pll_rate = 129; /* (256 - 128) + 1 */
428          /* slow to 30 MHz */
429          while(tmgptr->pll_locked == 1)
430          {
431              if(!--maxtries)
432              {
433                  ++failures;
434                  failedthistime = TRUE;
435                  break;
436              }
437          }
438          if(failures >= 100)
439          {
440              CPU_ENABLE_INTERRUPTS;
441              break; /* break out of loop */
442          }
443          if(failedthistime == TRUE)
444          {
445              CPU_ENABLE_INTERRUPTS;
446              continue; /* go try again */
447          }
448          else
449          {
450              ++successes; /* must have been a success */
451              successsthistime = TRUE;
452          }
453      }
454      debounced = 0;
455      maxtries = 10000;
456      while(!debounced) /* wait for lock and debounce, too */
457      {
458          for(i = 0; i < 50; i++) /* look for 50 samples in row */
459          {
460              if(!(--maxtries))
461              {
462                  CPU_ENABLE_INTERRUPTS;
463                  (void)lm_error("PLL Lock Time: Timeout waiting for lock true(2)\n");
464                  return FAILURE;
465              }
466              if(!tmgptr->pll_locked)
467                  break;
468              if(i == 0) /* get the time of first lock */
469              {
470                  tmgptr->ctc_register[3].value = LATCH_CNTR_1;
471                  for(i = 0; i < 3; i++)
472                      ctrl[i] = tmgptr->ctc_register[i].value & 0x0ff;
473                  if(ctrl[0] & 0x40) /* if null flag set */
474                      value = 01;
475                  else
476                  {
477                      value = ctrl[1] + (ctrl[2] << 8);
478                      if(value == 01)
479                          value = 11;
480

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_diag.c

DATE 5/23/89
TIME 4:41:33 pm

PAGE #
5/181

```

LINE # SOURCE TEXT
481     else
482         value = success1 - value;
483     }
484 }
485
486 if(i == 50)
487     debounce = 1;
488 }
489 CPU_ENABLE_INTERRUPTS;
490 if(successstatetime == TRUE)
491     slowdowntime += value;
492 } while(successes < 10);
493 if(successes == 0)
494     dowarreturncode = FAILURE;
495 }
496 else
497     slowdowntime = slowdowntime + 2667 / (10000 * successes);
498     /* message("slow down time = %d usecs\n", slowdowntime); */
499 }
500 usecs = (slowuptime > slowdowntime) ? slowuptime : slowdowntime;
501 *timeptr = usecs;
502 if(upreturncode == SUCCESS) || (dowarreturncode == SUCCESS)
503     return SUCCESS;
504 else
505     return FAILURE;
506 }
507
508 // long tmg_pll_lock(void)
509 // This function checks PLL lock time against limits. It
510 // calls tmg_plltime() to measure the time.
511 // Inputs: none
512 // Outputs: function returns SUCCESS or FAILURE
513
514 tmg_pll_lock()
515 {
516     u_long locktime;
517     if(tmg_plltime(&locktime) != SUCCESS)
518     {
519         (void)lm_error("PLL Lock Time: Error measuring lock time.\n");
520         return(FAILURE);
521     }
522     (void)lm_message("PLL Lock Time is %ld microseconds.\n", locktime);
523     if(locktime > 1500)
524     {
525         lm_error("PLL Lock Time (%d usec) exceeds 1500 usec bounds.\n");
526         return FAILURE;
527     }
528     else
529         return(SUCCESS);
530 }
531
532 // long tmg_pll_rate(void)
533 // This routine checks the PLL frequencies for various
534 // values of N to verify all aspects of divide-by-N counter
535 // Counter load values will be: 1 (N = 256)
536 // 128 (N = 128)
537 // 55a (N = 172)
538 // 2ah (N = 42)
539
540 Input: nothing
541 Return: SUCCESS or FAILURE
542
543 Test failures are those where tmg_checkrate() returns
544 discrepancy of more than 1 clock.
545
546 PLL freq: fo = 2 * N * 30 MHz / 256
547 TTL0800V CLK freq: TTL0800V_CLK / 8
548 TTL0800V CLK freq: either fo/2, fo/2K, or fo/4K
549 TTL0800V12 freq: 30 MHz / 8 (reference to 2254)
550
551 The 2254 counts the number of reference clocks (TTL0800V12)
552 that occur within 1000 unknown clocks (TTL0800V). This results
553 in a measurement that is accurate to at least 1 part in 1000
554 for the unknown frequency.
555
556 The time for 1000 clocks of unknown frequency is found to be
557 t = 1000 / ((2 * (N * (30 MHz / 256) / 16) / X) * X - 2 * 2K, or 4K
558
559 The number of reference clocks that expire in an interval t is
560 nlocks = t * (30 MHz / 8)
561 = 1000 / ((N / 256) / X) * X - 2 * 2K, or 4K
562
563 Even if the PLL were perfect, there could be +/- 1 clock in our
564 measurement. Some of the calculated number of clocks are non-integers
565 that with a little error on the PLL, could result in another clock
566 being measured. Also, depending on how many of the PLL reference
567 periods (117 KHz) occur within the measurement interval, can get
568 more error. It is reasonable to expect 0.1% error over the short
569 haul.
570
571 // tmg_pll_rate()
572 {
573     int i;
574     int returncode;
575     int nlocks;
576     /* N = 256 => fo = 60 MHz, should count 3000 clocks */
577     /* N = 128 => fo = 30 MHz, should count 1500 clocks */
578     /* N = 172 => fo = 40.3125 MHz, should count 2037.6 clocks */
579     /* N = 215 => fo = 50.390625 MHz, should count 2519.0 clocks */
580     static int n[] = { 256, 128, 172, 215 };
581     static long clocks[] = { 3000, 1500, 2037.6, 2519.0 };
582     static int delta[] = { 3, 5, 5, 4 };
583     static char error_format[] =
584         "%s Divide by %d failure. %d Expected %d clocks, measured %d clocks\n";
585     returncode = SUCCESS;
586     for (i = 0; i < 4; ++i) {
587         if(tmg_checkrate(n[i], &nlocks, &delta[i]) != SUCCESS)
588             returncode = FAILURE;
589     }
590 }

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_diag.c | DATE 5/23/89 | PAGE # 6/182 |
|--|---|------------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 601 | returncode = FAILURE; | | | |
| 602 | if (lm_error("PLL Div by N: tmg_checkrate() returned error.\n"); | | | |
| 603 | != SUCCESS) return FAILURE; | | | |
| 604 | else | | | |
| 605 | { | | | |
| 606 | if(abc(soclocks) > Delta[i]) /* 10 + 1 */ | | | |
| 607 | { | | | |
| 608 | returncode = FAILURE; | | | |
| 609 | if (lm_error(error_format, "PLL Div by N:", | | | |
| 610 | s[i], clocks[i], clocks[i] + soclocks) != SUCCESS) | | | |
| 611 | return FAILURE; | | | |
| 612 | } | | | |
| 613 | } | | | |
| 614 | return(returncode); | | | |
| 615 | } | | | |
| 616 | /* | | | |
| 617 | long tmg_pll_div(void) | | | |
| 618 | { | | | |
| 619 | /* this test checks divide by 2K counter and last div 4K flip flop | | | |
| 620 | /* All tests run using T= 60 MHz | | | |
| 621 | /* Input: nothing | | | |
| 622 | /* Returns: SUCCESS or FAILURE | | | |
| 623 | */ | | | |
| 624 | tmg_pll_div() | | | |
| 625 | { | | | |
| 626 | int returncode; | | | |
| 627 | int error; | | | |
| 628 | returncode = SUCCESS; | | | |
| 629 | /* 2K, K=2, fclocks = 1000 / ((K/256)/2K) * N = 256 | | | |
| 630 | if(tmg_checkrate(256,4,40001,error) != SUCCESS) | | | |
| 631 | return(FAILURE); | | | |
| 632 | else | | | |
| 633 | if(abc(error) > 5) /* 10 + 1 */ | | | |
| 634 | { | | | |
| 635 | (void)lm_error("PLL Div by K: Divide by 2 failure.\n"); | | | |
| 636 | returncode = FAILURE; | | | |
| 637 | } | | | |
| 638 | /* 2K, K = 256 | | | |
| 639 | if(tmg_checkrate(256,512,5120001,error) != SUCCESS) | | | |
| 640 | return(FAILURE); | | | |
| 641 | else | | | |
| 642 | if(abc(error) > 5) /* can't believe it could be that high */ | | | |
| 643 | { | | | |
| 644 | (void)lm_error("PLL Div by K: Divide by 256 failure.\n"); | | | |
| 645 | returncode = FAILURE; | | | |
| 646 | } | | | |
| 647 | /* 2K, K = 172 (pll_divisor = 8x35) | | | |
| 648 | if(tmg_checkrate(256,344,3440001,error) != SUCCESS) | | | |
| 649 | return(FAILURE); | | | |
| 650 | else | | | |
| 651 | if(abc(error) > 5) /* same as above */ | | | |
| 652 | { | | | |
| 653 | (void)lm_error("PLL Div by K: Divide by 172 failure.\n"); | | | |
| 654 | returncode = FAILURE; | | | |
| 655 | } | | | |
| 656 | /* 2K, K = 87 (pll_divisor = 8x88) | | | |
| 657 | if(tmg_checkrate(256,174,1740001,error) != SUCCESS) | | | |
| 658 | return(FAILURE); | | | |
| 659 | else | | | |
| 660 | if(abc(error) > 5) /* same as above */ | | | |
| 661 | { | | | |
| 662 | (void)lm_error("PLL Div by K: Divide by 87 failure.\n"); | | | |
| 663 | returncode = FAILURE; | | | |
| 664 | } | | | |
| 665 | return(returncode); | | | |
| 666 | /* | | | |
| 667 | tmg_slow_detect(void) | | | |
| 668 | { | | | |
| 669 | /* checks slow clock detector | | | |
| 670 | /* Inputs: none | | | |
| 671 | /* Returns: SUCCESS or FAILURE | | | |
| 672 | */ | | | |
| 673 | tmg_slow_detect() | | | |
| 674 | { | | | |
| 675 | int returncode; | | | |
| 676 | u_long start; | | | |
| 677 | returncode = SUCCESS; | | | |
| 678 | if(tmg_clockoff() != SUCCESS) | | | |
| 679 | { | | | |
| 680 | (void)lm_error("Slow Clock Detect: Could not turn clock off.\n"); | | | |
| 681 | return(FAILURE); | | | |
| 682 | } | | | |
| 683 | tmgptr->slow_clock_clear1 = 0; | | | |
| 684 | if(tmgptr->slow_clock) | | | |
| 685 | { | | | |
| 686 | (void)lm_error("Slow Clock Detect: Slow status bit stuck true.\n"); | | | |
| 687 | tmgptr->slow_clock_clear1 = 0; | | | |
| 688 | return(FAILURE); | | | |
| 689 | } | | | |
| 690 | tmgptr->pll_rate = 1; /* PLL set to 60 MHz */ | | | |
| 691 | tmgptr->pll_divisor = 33; /* K = 224 (134 KHz) */ | | | |
| 692 | tmgptr->clock_select = 1; /* Select divide by 2K */ | | | |
| 693 | lm_delay(10); /* wait for lock */ | | | |
| 694 | if(tmg_clockon() != SUCCESS) | | | |
| 695 | { | | | |
| 696 | (void)lm_error("Slow Clock Detect: cannot turn clock on.\n"); | | | |
| 697 | return(FAILURE); | | | |
| 698 | } | | | |
| 699 | start = lm_time(); | | | |
| 700 | do | | | |
| 701 | { | | | |
| 702 | if((lm_time() - start) > TIMEOUT) | | | |
| 703 | { | | | |
| 704 | (void)lm_error("Slow Clock Detect: timeout trying to initialize.\n"); | | | |
| 705 | return FAILURE; | | | |
| 706 | } | | | |
| 707 | tmgptr->slow_clock_clear1 = 0; /* clear it */ | | | |
| 708 | tmgptr->slow_clock_clear1 = 1; /* unclear it */ | | | |
| 709 | } | | | |
| 710 | } | | | |
| 711 | } | | | |
| 712 | } | | | |
| 713 | } | | | |
| 714 | } | | | |
| 715 | } | | | |
| 716 | } | | | |
| 717 | } | | | |
| 718 | } | | | |
| 719 | } | | | |
| 720 | } | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_diag.c

DATE 5/23/89 PAGE #
TIME 4:41:33 pm 7/183

```

LINE # SOURCE TEXT
721 } while(tmgptr->slow_clock), /* it set, repeat */
722 /* now the flip flop is really cleared, lets see if it sets */
723 lm_delay(10); /* detector should not fire */
724 /* is about 8 usecs */
725 if(tmgptr->slow_clock)
726 {
727     (void)lm_error("Slow Clock Detect: detector fired at 134 KHz.\n");
728     returncode = FAILURE;
729 }
730 if(tmg_clockoff() != SUCCESS)
731 {
732     (void)lm_error("Slow Clock Detect: cannot turn clock off.\n");
733     return(FAILURE);
734 }
735 tmgptr->slow_clock_clear1 = 0;
736 tmgptr->pll_divisor = 1; /* K = 256 (117 KHz) */
737 if(tmg_clockon() != SUCCESS)
738 {
739     (void)lm_error("Slow Clock Detect: cannot turn clock on.\n");
740     return(FAILURE);
741 }
742 lm_delay(1); /* wait for freq to stabilize */
743 tmgptr->slow_clock_clear1 = 1;
744 lm_delay(10); /* detector should fire */
745 /* is about 8 usecs */
746 if(!tmgptr->slow_clock) /* should be set */
747 {
748     (void)lm_error("Slow Clock Detect: detector did not fire at 117 KHz.\n");
749     returncode = FAILURE;
750 }
751 if(tmg_clockoff() != SUCCESS)
752 {
753     (void)lm_error("Slow Clock Detect: cannot turn clock off.\n");
754     return(FAILURE);
755 }
756 tmgptr->pll_divisor = 33; /* K = 324 (134 KHz) */
757 lm_delay(1);
758 if(tmg_clockon() != SUCCESS)
759 {
760     (void)lm_error("Slow Clock Detect: cannot turn clock on.\n");
761     return(FAILURE);
762 }
763 if(!tmgptr->slow_clock) /* should still be set */
764 {
765     (void)lm_error("Slow Clock Detect: output cleared without command.\n");
766     returncode = FAILURE;
767 }
768 tmgptr->slow_clock_clear1 = 0;
769 return(returncode);
770 }
771
772 /*
773 * int tmg_freq_ext0(void)
774 * This function measures the frequency of External Clock 0.
775 * Inputs: none
776 * Outputs: function returns SUCCESS or FAILURE
777 */
778 int tmg_freq_ext0()
779 {
780     unsigned long hertz, noclocks;
781     if(tmg_clockoff() != SUCCESS)
782     {
783         (void)lm_error("Measure Ext0 Frequency: cannot shut clock off.\n");
784         return(FAILURE);
785     }
786     tmgptr->clock_select = 3;
787     if(tmg_measure_freq(&noclocks) != SUCCESS)
788     {
789         (void)lm_error("Measure Ext0 Frequency: FAILURE!\n");
790         return(FAILURE);
791     }
792     hertz = (60ll / (double)noclocks);
793     (void)lm_banner("External Clock 0 frequency is %ld Hz.\n", hertz);
794     return(SUCCESS);
795 }
796
797 /*
798 * int tmg_freq_ext1(void)
799 * This function measures the frequency of External Clock 1.
800 * Inputs: none
801 * Outputs: function returns SUCCESS or FAILURE
802 */
803 int tmg_freq_ext1()
804 {
805     unsigned long hertz, noclocks;
806     if(tmg_clockoff() != SUCCESS)
807     {
808         (void)lm_error("Measure Ext1 Frequency: cannot shut clock off.\n");
809         return(FAILURE);
810     }
811     tmgptr->clock_select = 4;
812     if(tmg_measure_freq(&noclocks) != SUCCESS)
813     {
814         (void)lm_error("Measure Ext1 Frequency: FAILURE!\n");
815         return(FAILURE);
816     }
817     hertz = (60ll / (double)noclocks);
818     (void)lm_banner("External Clock 1 frequency is %ld Hz.\n", hertz);
819     return(SUCCESS);
820 }
821
822 /*
823 * long tmg_interrupts(void)
824 * This function tests Timing Generator Interrupt capability.
825 * Inputs: none
826 * Outputs: function returns SUCCESS or FAILURE
827 */
828 long tmg_interrupts()
829 {
830     long returncode;
831     unsigned long start;

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_diag.c

DATE 5/23/89
TIME 4:41:33 pm

PAGE #
8/184

```

LINE #          SOURCE TEXT
841  int zero = 0;
842
843  returncode = SUCCESS;
844
845  if(tmg_reset(FALSE) != SUCCESS)
846  {
847      (void)lm_error("Interrupt Test: cannot reset.\n");
848      return(FAILURE);
849  }
850
851  if(tmg_clockoff() != SUCCESS)
852  {
853      (void)lm_error("Interrupt Test: cannot turn clock off.\n");
854      if(tmg_clockon() != SUCCESS)
855      {
856          (void)lm_error("Interrupt Test: cannot turn clock on.\n");
857          return(FAILURE);
858      }
859  }
860  /* first check that no interrupts exist */
861  if(TMG_INT)
862  {
863      (void)lm_error("Interrupt Test: interrupt active after reset, check CPU.\n");
864      return(FAILURE);
865  }
866  /* check CTC Interrupt Capability */
867  (void)lm_message("Checking CTC Interrupt\n");
868  if(tmgptr->ctc_intr)
869  {
870      (void)lm_error("Interrupt Test: CTC Interrupt active after reset.\n");
871      returncode = FAILURE;
872  }
873
874  tmgptr->ctc_register[3].value = CTR0CN; /* initialize ctr.0 */
875  tmgptr->ctc_intr_enable = 1;
876  tmgptr->ctc_register[0].value = 1; /* lab of very short count */
877  tmgptr->ctc_register[0].value = zero; /* sub of very short count */
878  tmgptr->ctc_intr_clear = 1;
879  start = lm_time(); /* start timer */
880  while(!TMG_INT)
881  {
882      if((lm_time() - start) > 10)
883      {
884          (void)lm_error("Interrupt Test: CTC failed to interrupt.\n");
885          returncode = FAILURE;
886          break;
887      }
888      if(!tmgptr->ctc_intr)
889      {
890          (void)lm_error("Interrupt Test: expected CTC interrupt, received other.\n");
891          returncode = FAILURE;
892      }
893      else
894      {
895          (void)lm_message("Interrupt Test: CTC interrupt received OK.\n");
896          tmgptr->ctc_intr_clear = 0; /* clear output */
897          tmgptr->ctc_intr_enable = 0; /* disable interrupt */
898      }
899  }
900  /* now check EOP Interrupt Capability */
901  (void)lm_message("Checking End-of-Play Interrupt\n");
902  tmg_ptr->test_mode[1];
903  tmgptr->pattern_intr_enable = 1; /* enable EOP interrupt */
904  tmgptr->start_pattern_play = 1; /* start the PLAY */
905  start = lm_time();
906  while(!TMG_INT)
907  {
908      if((lm_time() - start) > 10)
909      {
910          (void)lm_error("Interrupt Test: EOP failed to interrupt.\n");
911          returncode = FAILURE;
912          break;
913      }
914      if(TMG_INT)
915      {
916          (void)lm_message("Interrupt Test: EOP interrupt received OK.\n");
917          tmgptr->pattern_intr_enable = 0;
918          if(TMG_INT)
919          {
920              (void)lm_error("Interrupt Test: EOP interrupt did not clear.\n");
921              returncode = FAILURE;
922          }
923      }
924      /* finally check Error Interrupt Capability */
925      (void)lm_message("Checking Backplane Error Interrupt\n");
926      if(tmg_clockoff() != SUCCESS)
927      {
928          (void)lm_warning("Interrupt Test: cannot turn clock off.\n");
929      }
930      if(tmg_clockon() != SUCCESS)
931      {
932          (void)lm_warning("Interrupt Test: cannot turn clock on.\n");
933      }
934      tmgptr->backplane_intr_enable = 1;
935      if(TMG_INT)
936      {
937          (void)lm_error("Interrupt Test: backplane is driving ERROR.\n");
938      }
939      else
940      {
941          tmgptr->backplane_error = 1;
942          start = lm_time();
943          while(!TMG_INT)
944          {
945              if((lm_time() - start) > 10)
946              {
947                  (void)lm_error("Interrupt Test: Unable to cause BP Error Interrupt.\n");
948                  returncode = FAILURE;
949                  break;
950              }
951          }
952          if(TMG_INT)
953          {
954              (void)lm_message("Interrupt Test: Backplane Error Interrupt received OK.\n");
955              tmgptr->backplane_error = 0;
956              tmgptr->backplane_intr_enable = 0;
957              if(TMG_INT)
958              {
959                  (void)lm_error("Interrupt Test: Backplane interrupt did not clear.\n");
960                  returncode = FAILURE;
961              }
962          }
963      }
964  }
965  return(returncode);
966
967  long tmg_loop(void)
968  {
969      /* This function is used for running continuous patterns. Actual
970      /* looping is handled by menu code.
971      /*
972      /* Inputs: none

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_diag.c

DATE 5/23/89

PAGE #

TIME 4:41:33 pm

9/185

```

LINE # SOURCE TEXT
961 /* Outputs: function returns SUCCESS or FAILURE
962 */
963 tmg_loop()
964 {
965     /* Make sure clock is off */
966     if (tmg_clockoff() != SUCCESS)
967         return(FAILURE);
968     if (tmg_saffclear() != SUCCESS)
969         (void)lm_warning("Loop: Cal flip flops did not clear.\n");
970     /* then turn it on */
971     if (tmg_clockon() != SUCCESS)
972         return(FAILURE);
973     if (tmg_play(TIMEOUT) != SUCCESS)
974         return(FAILURE);
975     return(SUCCESS);
976 }
977
978 /*
979  * int tmg_select_mode(void)
980  *
981  * This routine is a utility to allow setting of the desired
982  * test and sample modes from the diagnostic menus.
983  *
984  * Input: none
985  * Output: Always returns SUCCESS
986  */
987 tmg_select_mode()
988 {
989     u_long answer;
990     (void)lm_message("Select test mode:\n");
991     (void)lm_message("(t0) Normal Mode\n");
992     (void)lm_message("(t1) Test Mode 1 (1 edge ramp, 1 sample ramp) (default)\n");
993     (void)lm_message("(t2) Test Mode 2 (2 edge ramps, 1 sample ramp)\n");
994     (void)lm_message("(t3) Test Mode 3 (Continuous edge ramps)\n");
995     answer = 1; /* test mode 1 */
996     diag_get_ulong(&answer, "Choice", 01, 31);
997     tmgptr->test_mode = answer;
998     (void)lm_message("Select sample mode:\n");
999     (void)lm_message("(s0) Early sample\n");
1000     (void)lm_message("(s1) Edge triggered sample (default)\n");
1001     answer = 1;
1002     diag_get_ulong(&answer, "Choice", 01, 11);
1003     tmgptr->sample_mode = answer;
1004     return SUCCESS;
1005 }
1006
1007 /*
1008  * int tmg_select_ramps(void)
1009  *
1010  * Choose ramps from diagnostic menus.
1011  *
1012  * Input: none
1013  * Output: Always returns SUCCESS
1014  */
1015 tmg_select_ramps()
1016 {
1017     u_long answer;
1018     lm_message("Select ramps:\n");
1019     answer = 0;
1020     diag_get_ulong(&answer, "edge ramp", 01, 31);
1021     tmgptr->edge_delay_range = answer;
1022     answer = 0;
1023     diag_get_ulong(&answer, "sample ramp", 01, 31);
1024     tmgptr->sample_delay_range = answer;
1025     return SUCCESS;
1026 }
1027
1028 /*
1029  * int tmg_select_thresholds(void)
1030  *
1031  * This routine allows setting edge/sample thresholds from
1032  * the diagnostic menus.
1033  *
1034  * Input: none
1035  * Output: always returns SUCCESS
1036  */
1037 tmg_select_thresholds()
1038 {
1039     u_long answer;
1040     char buffer[32];
1041     int i;
1042     (void)lm_message("Select thresholds:\n");
1043     for(i = 0; i < 6; i++)
1044     {
1045         answer = 10;
1046         sprintf(buffer, "edge[%d] threshold", i);
1047         diag_get_ulong(&answer, buffer, 01, 2551);
1048         tmgptr->edge_delay[i].delay = answer;
1049     }
1050     answer = 10;
1051     diag_get_ulong(&answer, "sample trigger threshold", 01, 2551);
1052     tmgptr->sample_trigger_threshold = answer;
1053     answer = 10;
1054     diag_get_ulong(&answer, "sample threshold", 01, 255);
1055     tmgptr->sample_delay = answer;
1056     return SUCCESS;
1057 }
1058
1059 /*
1060  * int tmg_select_slot_count(void)
1061  *
1062  * This utility allows setting slot count from diagnostic menus.
1063  *
1064  * Input: none
1065  * Output: always returns SUCCESS
1066  */
1067 tmg_select_slot_count()
1068 {
1069     u_long answer;
1070     answer = 1;
1071     diag_get_ulong(&answer, "slot count", 11, 81);
1072     tmg_set_slot_count(answer);
1073     return SUCCESS;
1074 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_diag.c

DATE 5/23/89 PAGE #
TIME 4:41:33 pm 10/186

```

1081 }
1082
1083
1084
1085 /*
1086  * int tmg_edge_jitter(void)
1087  *
1088  * This utility sets up all edges at maximum threshold and allows the
1089  * operator to step through the four ramps. This makes taking an
1090  * oscilloscope measurement of the jitter as painless as possible.
1091  *
1092  * Inputs: none
1093  * Outputs: always returns SUCCESS
1094  */
1095 tmg_edge_jitter()
1096 {
1097     int ramp, edge, period;
1098     static int jitter[] = {500, 1000, 2000, 10000, 50000};
1099     lm_message("Edge Jitter Measurement\n");
1100     lm_message("Trigger the oscilloscope on rising edge of U130-15, (CECLISE_CHARGE).\n");
1101     lm_message("Jitter is the maximum uncertainty in position of the edge rising edge.\n");
1102     tmg_set_slot_count(2);
1103     tmg_set_test_mode(1);
1104     for(edge = 0; edge < 6; edge++)
1105     {
1106         tmgptr->edge_delay[edge].delay = 250;
1107         for(ramp = 0; ramp < 4; ramp++)
1108         {
1109             tmgptr->edge_delay_range = ramp;
1110             period = calib.EdgeMaxDelay[ramp] * 1.5;
1111             tmg_set_period(ramp, tperiod);
1112             lm_message("Selected edge ramp is %d\n", ramp);
1113             lm_message("Verify jitter to be less than %d ps for all 6 edges.\n",
1114                 jitter[ramp]);
1115             tmg_play_til_key();
1116         }
1117     }
1118     return SUCCESS;
1119 }
1120
1121 /*
1122  * int tmg_sample_jitter(void)
1123  *
1124  * This utility sets up the sample at maximum threshold and allows the
1125  * operator to step through the four ramps in both early and edge 7
1126  * triggered modes. This makes taking an
1127  * oscilloscope measurement of the jitter as painless as possible.
1128  *
1129  * Inputs: none
1130  * Outputs: always returns SUCCESS
1131  */
1132 tmg_sample_jitter()
1133 {
1134     int ramp, edge, period;
1135     static int jitter[] = {500, 1000, 2000, 4000, 500, 2000, 10000, 50000};
1136     lm_message("Sample Jitter Measurement\n");
1137     lm_message("Trigger the oscilloscope on rising edge of U130-15, (CECLISE_CHARGE).\n");
1138     lm_message("Jitter is the maximum uncertainty in position of the sample rising edge.\n");
1139     lm_message("Early trigger mode:\n");
1140     tmg_set_slot_count(2);
1141     tmg_set_test_mode(1);
1142     tmgptr->sample_delay = 250;
1143     tmgptr->sample_mode = 0;
1144     period = 150000;
1145     for(ramp = 0; ramp < 4; ramp++)
1146     {
1147         tmgptr->sample_delay_range = ramp;
1148         tmg_set_period(ramp, tperiod);
1149         lm_message("Selected sample ramp is %d\n", ramp);
1150         lm_message("Verify jitter to be less than %d ps.\n",
1151             jitter[ramp]);
1152         tmg_play_til_key();
1153     }
1154     lm_message("Edge 7 triggered mode:\n");
1155     tmg_set_slot_count(2);
1156     tmg_set_test_mode(1);
1157     tmgptr->sample_trigger_threshold = 250;
1158     tmgptr->sample_mode = 1;
1159     tmgptr->sample_delay_range = 0;
1160     tmgptr->sample_delay = 10;
1161     for(ramp = 0; ramp < 4; ramp++)
1162     {
1163         tmgptr->edge_delay_range = ramp;
1164         period = calib.EdgeMaxDelay[ramp] * 1.5;
1165         tmg_set_period(ramp, tperiod);
1166         lm_message("Selected edge ramp is %d\n", ramp);
1167         lm_message("Verify jitter to be less than %d ps.\n",
1168             jitter[ramp * 4]);
1169         tmg_play_til_key();
1170     }
1171     return SUCCESS;
1172 }
1173
1174 /*
1175  * int tmg_edge_alignment(void)
1176  *
1177  * This test sets up all edges to its best approximation of 50 ns
1178  * delay. The clock period is set to 100 ns. This allows the operator
1179  * to confirm edge alignment with an oscilloscope.
1180  */
1181 tmg_edge_alignment()
1182 {
1183     int edge;
1184     int period;
1185     period = 100000; /* 100000 ps */
1186     lm_message("Edge alignment test:\n");
1187     if(!calib.CalCompleted)
1188     {
1189         lm_message("Please wait while calibration is performed...\n");
1190         tmg_calibrate();
1191         if(!calib.CalCompleted)
1192         {
1193             lm_message("This test requires that calibration be successfully completed.\n");
1194             return SUCCESS;
1195         }
1196     }
1197     else lm_message("\n\n");
1198     lm_message("Target edge position is 50 ns with a clock period of 100 ns\n");
1199     lm_message("A convenient trigger is U130-15 (CECLISE_CHARGE).\n");
1200     tmg_set_slot_count(1);
1201     tmg_set_test_mode(1);

```

| | | | | |
|--|------------------------------------|------|------------|--------|
| Copyright 1989 Logic Modeling Systems | SOURCE PROGRAM diags/tmg_diag.c | DATE | 5/23/89 | PAGE # |
| | | TIME | 4:41:33 pm | 11/187 |

| LINE # | SOURCE TEXT |
|--------|---|
| 1201 | tmgptr->edge_delay_range = 0; |
| 1202 | tmgptr->sample_delay_range = 0; |
| 1203 | for(edge = 0; edge < 6; edge++) |
| 1204 | tmgptr->edge_delay[edge].delay = (500001 - (long)calib.EdgeOffset[0][edge]) |
| 1205 | / (long)calib.EdgeSlope[0][edge]; |
| 1206 | tmg_set_period(0, tperiod); /* temp 0, 100 ns */ |
| 1207 | lm_message("Verify maximum skew between edges is 1000 ps max.\n"); |
| 1208 | lm_message("Check all edges of all lanes of slot %d\n", |
| 1209 | tmg_play_til_key()); |
| 1210 | |
| 1211 | |
| 1212 | /* long tmg_continuousclocktest(void) |
| 1213 | /* |
| 1214 | This routine is a utility to allow measuring the PLL lock |
| 1215 | time on an oscilloscope or to diagnose the PLL. The PLL |
| 1216 | is continuously slowed from 30 to 60 MHz and back. |
| 1217 | Inputs: none |
| 1218 | Outputs: after a key is pressed, always returns SUCCESS |
| 1219 | */ |
| 1220 | tmg_continuousclocktest() |
| 1221 | { |
| 1222 | if(tmg_clockoff() != SUCCESS) /* make sure clock is off */ |
| 1223 | { |
| 1224 | (void)lm_error("tmg_continuousclocktest(): cannot turn clock off\n"); |
| 1225 | return(FAILURE); |
| 1226 | } |
| 1227 | |
| 1228 | tmgptr->pll_rate = (256 - 128) + 1; /* sets to 30 MHz */ |
| 1229 | tmgptr->pll_divisor = (256 - 15) + 1; /* choose div 2 = 15 */ |
| 1230 | tmgptr->clock_select = 1; /* sel div 2K */ |
| 1231 | lm_delay(2); /* wait lots of time for lock */ |
| 1232 | (void)lm_message("You may observe TTLLOCK at U27-13 while triggering.\n"); |
| 1233 | (void)lm_message("the oscilloscope with TRIGRATE(7) at U160-10.\n"); |
| 1234 | (void)lm_message("Press any key to abort...\n"); |
| 1235 | while(!lm_check_key()) |
| 1236 | { |
| 1237 | tmgptr->pll_rate = (256 - 256) + 1; /* set pll to 60 MHz */ |
| 1238 | lm_delay(2); /* wait for lock */ |
| 1239 | tmgptr->pll_rate = (256 - 128) + 1; /* set pll to 30 MHz */ |
| 1240 | lm_delay(2); |
| 1241 | } |
| 1242 | return SUCCESS; |
| 1243 | } |
| 1244 | |
| 1245 | /* int tmg_clock_sync_test() |
| 1246 | /* |
| 1247 | This test checks that the synchronizer really can turn the clock |
| 1248 | on and off. It returns SUCCESS or FAILURE. |
| 1249 | */ |
| 1250 | int tmg_clock_sync_test() |
| 1251 | { |
| 1252 | long start; |
| 1253 | static char me[] = "tmg_clock_sync_test"; |
| 1254 | tmgptr->clock_enable = 0; /* try to turn the clock off */ |
| 1255 | start = lm_time(); |
| 1256 | while(tmgptr->clock_on) |
| 1257 | { |
| 1258 | if((lm_time() - start) > 10) |
| 1259 | { |
| 1260 | if(tmgptr->clock_select > 2) /* must be external clock */ |
| 1261 | tmgptr->clock_sync_clearl = 0; /* ok to clear sync */ |
| 1262 | else |
| 1263 | { |
| 1264 | lm_error("%s: cannot clear synchronizer\n", me); |
| 1265 | return FAILURE; |
| 1266 | } |
| 1267 | } |
| 1268 | } |
| 1269 | tmgptr->clock_sync_clearl = 0; /* keep it cleared */ |
| 1270 | tmgptr->clock_select = 0; |
| 1271 | tmgptr->pll_rate = 0x1c; /* set up for 15 MHz */ |
| 1272 | tmgptr->pll_divisor = 0x1f; /* divide by 2 */ |
| 1273 | tmgptr->clock_enable = 1; /* hope it doesn't come on */ |
| 1274 | start = lm_time(); |
| 1275 | while((lm_time() - start) > 10) |
| 1276 | { |
| 1277 | if(tmgptr->clock_on) /* clock came on, bad clock */ |
| 1278 | return FAILURE; |
| 1279 | } |
| 1280 | tmgptr->clock_enable = 0; |
| 1281 | tmgptr->clock_sync_clearl = 1; |
| 1282 | start = lm_time(); |
| 1283 | while((lm_time() - start) > 10) |
| 1284 | { |
| 1285 | if(tmgptr->clock_on) /* clock came on, bad clock */ |
| 1286 | return FAILURE; |
| 1287 | } |
| 1288 | tmgptr->clock_enable = 1; /* this time it should come on */ |
| 1289 | start = lm_time(); |
| 1290 | while(!tmgptr->clock_on) |
| 1291 | { |
| 1292 | if((lm_time() - start) > 10) |
| 1293 | { |
| 1294 | lm_error("%s: timeout waiting for clock to turn on.\n", me); |
| 1295 | return FAILURE; |
| 1296 | } |
| 1297 | } |
| 1298 | tmgptr->clock_enable = 0; /* this time it should go off */ |
| 1299 | start = lm_time(); |
| 1300 | while(tmgptr->clock_on) |
| 1301 | { |
| 1302 | if((lm_time() - start) > 10) |
| 1303 | { |
| 1304 | lm_error("%s: timeout waiting for clock to turn on.\n", me); |
| 1305 | return FAILURE; |
| 1306 | } |
| 1307 | } |
| 1308 | tmgptr->clock_sync_clearl = 0; /* clear things and exit */ |
| 1309 | return SUCCESS; |
| 1310 | } |
| 1311 | |
| 1312 | /* |
| 1313 | int tmg_dac_settle(void) |
| 1314 | /* |
| 1315 | This routine slows DAC 0 output back and forth from 0 to 255 |
| 1316 | to allow measurement of slew rate and settling time on an |
| 1317 | */ |
| 1318 | |
| 1319 | |
| 1320 | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_diag.c

DATE

5/23/89

PAGE #

TIME

4:41:33 pm

12/188

```

1321 *      oscilloscope.
1322 *
1323 *      Input: none
1324 *      Output: always returns SUCCESS
1325 */
1326 tmg_dac_settle()
1327 {
1328     #define DAC (char *)0x00000001
1329     (void)lm_message("You may now observe ALOCISDA0 at U61-9.\n");
1330     (void)lm_message("Press any key to abort...\n");
1331     while(!lm_check_key())
1332     {
1333         *DAC = 0;
1334         lm_delay(2);
1335         *DAC = 255;
1336         lm_delay(2);
1337     }
1338 }
1339 return SUCCESS;
1340 }
1341
1342 #ifndef BIN_LIMIT
1343 #define BIN_LIMIT 1000
1344 #endif
1345
1346 tmg_edge_jitter_search(ramp,edge,thr,upper_bound,lower_bound,direction)
1347 long ramp, edge, thr, upper_bound, lower_bound, direction;
1348 {
1349     static char me[] = "tmg_edge_jitter_search";
1350     long period, found_high;
1351     long timeout = BIN_LIMIT;
1352     long chk_period, save_lower_bound, save_upper_bound;
1353     long jitterptr, sptr, lptr, selectptr;
1354
1355     save_lower_bound = lower_bound;
1356     save_upper_bound = upper_bound;
1357     for (timeout = 0; timeout <= BIN_LIMIT; ++timeout) {
1358         period = (upper_bound + lower_bound)/2;
1359         if (tmg_detect_edge(ramp,edge,
1360             (direction ? DETECT_LOW : 0) | DETECT_BOOL, &period,
1361             thr,10000L, &found_high) != SUCCESS) {
1362             tmg_report_failure(me, "tmg_detect_always_low(3)");
1363         }
1364         #ifdef DIAGS
1365             if (tmg_cal_debug & 1)
1366                 tmg_play_til_key();
1367         #endif
1368         if ((period < save_lower_bound) || (period > save_upper_bound))
1369             lm_error("%s binary search failure(1)\n", me);
1370         return period;
1371     }
1372
1373     if ((period <= lower_bound) || (period >= upper_bound)) {
1374         /* XXX */
1375         while (1) {
1376             tmg_get_next_lower_period(period, &chk_period,
1377                 &jitterptr,&sptr,&lptr,&selectptr);
1378             if (chk_period <= lower_bound) break;
1379             if (tmg_detect_edge(ramp,edge,
1380                 (direction ? DETECT_LOW : 0) |
1381                 DETECT_BOOL, &chk_period,
1382                 thr,10000L, &found_high) != SUCCESS) {
1383                 tmg_report_failure(me, "tmg_detect_edge(4)");
1384             }
1385             if (direction ? !found_high : found_high) {
1386                 /* too short - raise lower bound */
1387                 lower_bound = chk_period;
1388                 break;
1389             } else {
1390                 /* too long - lower upper bound */
1391                 upper_bound = chk_period;
1392                 period = chk_period;
1393             }
1394         }
1395         /* We can't get more exact than this */
1396         #ifdef DIAGS
1397             if (tmg_cal_debug & 4)
1398                 lm_message("(got it between %d and %d)\n",
1399                     lower_bound, upper_bound);
1400         #endif
1401         return (direction ? upper_bound : chk_period);
1402     }
1403     if (direction ? !found_high : found_high) {
1404         /* too short - raise lower bound */
1405         lower_bound = period;
1406     } else {
1407         /* too long - lower upper bound */
1408         upper_bound = period;
1409     }
1410     lm_error("%s binary search failure(2)\n", me);
1411     return period;
1412 }
1413 /* tmg_edge_jitter_search */
1414
1415 tmg_sample_jitter_search(eramp,eramp,thr7,thr,upper_bound,lower_bound,direction)
1416 long eramp, eramp, thr7, thr, upper_bound, lower_bound, direction;
1417 {
1418     static char me[] = "tmg_sample_jitter_search";
1419     long period, found_high;
1420     long timeout = BIN_LIMIT;
1421     long chk_period, save_lower_bound, save_upper_bound;
1422     long jitterptr, sptr, lptr, selectptr;
1423
1424     save_lower_bound = lower_bound;
1425     save_upper_bound = upper_bound;
1426     for (timeout = 0; timeout <= BIN_LIMIT; ++timeout) {
1427         period = (upper_bound + lower_bound)/2;
1428         if (tmg_detect_sample(eramp,eramp,
1429             (direction ? DETECT_LOW : 0) | DETECT_SAMPLE,
1430             (long*)&period, thr7,thr,10000L, (long*)&found_high) != SUCCESS)
1431             tmg_report_failure(me,

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_diag.c

DATE 5/23/89

PAGE #

TIME 4:41:33 pm

13/189

```

1441         "tmg_detect_sample(3)");
1442     }
1443 #ifdef DIAGS
1444     if (tmg_cal_debug & 0x100)
1445         tmg_play_til_key();
1446 #endif DIAGS
1447     if ((period < save_lower_bound) || (period > save_upper_bound))
1448     {
1449         lm_error("ta binary search failure(1)\n", me);
1450         return period;
1451     }
1452     if ((period <= lower_bound) || (period >= upper_bound)) {
1453         /* XXX */
1454         while (1) {
1455             tmg_get_next_lower_period(period, &chk_period,
1456                                     &jitterptr, &mptr, &selectptr);
1457             if (chk_period <= lower_bound) break;
1458             if (tmg_detect_sample(aramp, aramp,
1459                                 (direction ? DETECT_LOW : 0) |
1460                                 DETECT_BOOL | DETECT_SAMPLE,
1461                                 (long*)&chk_period, &thr7, &thr8,
1462                                 10000L, (long*)&found_high)
1463                 != SUCCESS)
1464             {
1465                 tmg_report_failure(me,
1466                                   "tmg_detect_sample(4)");
1467                 if (direction ? found_high : found_high) {
1468                     /* too short - raise lower bound */
1469                     lower_bound = chk_period;
1470                     break;
1471                 } else {
1472                     /* too long - lower upper bound */
1473                     upper_bound = chk_period;
1474                 }
1475                 period = chk_period;
1476             }
1477             /* We can't get more exact than this */
1478         }
1479     }
1480 #ifdef DIAGS
1481     if (tmg_cal_debug & 0x400)
1482         lm_message("(got it between %d and %d)\n",
1483                   lower_bound, upper_bound);
1484 #endif DIAGS
1485     return (direction ? upper_bound : lower_bound);
1486 }
1487 if (direction ? found_high : found_high) {
1488     /* too short - raise lower bound */
1489     lower_bound = period;
1490 } else {
1491     /* too long - lower upper bound */
1492     upper_bound = period;
1493 }
1494 lm_error("ta binary search failure(2)\n", me);
1495 return period;
1496 }
1497 /* tmg_sample_jitter_search */
1498
1499 tmg_jitter_test()
1500 {
1501     int aramp, aramp, edge;
1502     long low_bound, high_bound, target, low_target, high_target;
1503     if (!calib.CalCompleted)
1504     {
1505         tmg_calibrate();
1506         if (!calib.CalCompleted)
1507         {
1508             lm_error("Cannot perform jitter test due to calibration failure\n",
1509                     me);
1510             return FAILURE;
1511         }
1512     }
1513     tmg_set_test_mode(1);
1514     tmg_set_slot_count(2);
1515     tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE; /* select Edge7 trigger */
1516     for (aramp = 0; aramp < NUMBER_OF_EDGES; aramp++)
1517     {
1518         /* check jitter on all edges */
1519         for (edge = 0; edge < NUMBER_OF_EDGES; edge++)
1520         {
1521             target = tmg_predict_period(250, calib.Edgeslope[aramp][edge],
1522                                         calib.EdgeOffset[aramp][edge]);
1523             low_target = 0.9 * target;
1524             high_target = 1.1 * target;
1525             low_bound = tmg_edge_jitter_search(aramp, edge, 250, high_target, low_target, 0);
1526             high_bound = tmg_edge_jitter_search(aramp, edge, 250, high_target, low_target, 1);
1527             lm_message("Ramp %d, Edge %d, target %d, found %d-%d, jitter %d\n",
1528                       aramp, edge, target, low_bound, high_bound, high_bound - low_bound);
1529         }
1530         /* do edge 7 */
1531         target = 250 * (long)calib.Edge7Slope[aramp]
1532                 + 10 * (long)calib.SampleSlope[0]
1533                 + (long)calib.SampleOffset[aramp][0];
1534         low_target = 0.9 * target;
1535         high_target = 1.1 * target;
1536         low_bound = tmg_sample_jitter_search(aramp, 0, 250, 10,
1537                                             high_target, low_target, 0);
1538         high_bound = tmg_sample_jitter_search(aramp, 0, 250, 10,
1539                                             high_target, low_target, 1);
1540         lm_message("Ramp %d, sample, target %d, found %d-%d, jitter %d\n", aramp,
1541                   target, low_bound, high_bound, high_bound - low_bound);
1542     }
1543     /* now do sample */
1544     tmgptr->sample_mode = EARLYSAMPLETRIGGERMODE; /* select early sample */
1545     for (aramp = 0; aramp < NUMBER_OF_EDGES; aramp++)
1546     {
1547         target = 255 * (long)calib.SampleSlope[aramp]
1548                 + (long)calib.EarlySampleOffset[aramp];
1549         low_target = 0.9 * target;
1550         high_target = 1.1 * target;
1551         low_bound = tmg_sample_jitter_search(1, aramp, 255, 255,
1552                                             high_target, low_target, 0);
1553         high_bound = tmg_sample_jitter_search(1, aramp, 255, 255,
1554                                             high_target, low_target, 1);
1555         lm_message("Ramp %d, sample, target %d, found %d-%d, jitter %d\n", aramp,
1556                   target, low_bound, high_bound, high_bound - low_bound);
1557     }
1558 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_diag.c

DATE 5/23/89 PAGE #
TIME 4:41:33 pm 14/190

```

1561 return SUCCESS,
1562 }
1563
1564
1565
1566
1567 /* This function sets the period to 128 ns. It takes each edge in turn
1568 * and places it at 50 ns. It then takes all other (2) edges that share
1569 * the same delay line module and sweeps them from 50 ns to 70 ns. If
1570 * any coupling is detected, the function fails.
1571 */
1572 tmg_delay_line_isolation()
1573 {
1574     int target_edge, errors, count;
1575     long period;
1576
1577     errors = 0;
1578     tmgptr->backplane_reset = 0; /* assert backplane reset */
1579     tmg_set_slot_count(1);
1580     tmg_set_test_mode(1);
1581     tmg_set_sweep(0);
1582     period = 128000L;
1583     if(tmg_set_period(0, period) != SUCCESS)
1584     {
1585         lm_error("tmg_delay_line_isolation: tmg_set_period()");
1586         return FAILURE;
1587     }
1588
1589     for(target_edge = 0; target_edge < 6; target_edge++)
1590     {
1591         tmgptr->edge_delay[target_edge].delay = tmg_predict_threshold(50000L,
1592             calib.EdgeSlope[0][target_edge],
1593             calib.EdgeOffset[0][target_edge]);
1594
1595         for(count = 1000; count > 0; count--)
1596             switch (target_edge)
1597             {
1598                 case 0:
1599                     tmg_sweep(target_edge, 1, &errors, &calib);
1600                     tmg_sweep(target_edge, 2, &errors, &calib);
1601                     break;
1602                 case 1:
1603                     tmg_sweep(target_edge, 0, &errors, &calib);
1604                     tmg_sweep(target_edge, 2, &errors, &calib);
1605                     break;
1606                 case 2:
1607                     tmg_sweep(target_edge, 0, &errors, &calib);
1608                     tmg_sweep(target_edge, 1, &errors, &calib);
1609                     break;
1610                 case 3:
1611                     tmg_sweep(target_edge, 4, &errors, &calib);
1612                     tmg_sweep(target_edge, 5, &errors, &calib);
1613                     break;
1614                 case 4:
1615                     tmg_sweep(target_edge, 3, &errors, &calib);
1616                     tmg_sweep(target_edge, 5, &errors, &calib);
1617                     break;
1618                 case 5:
1619                     tmg_sweep(target_edge, 3, &errors, &calib);
1620                     tmg_sweep(target_edge, 4, &errors, &calib);
1621                     break;
1622                 default:
1623                     break;
1624             }
1625
1626         (void)diag_tmg_reset(1); /* Full reset */
1627         if(errors)
1628             return FAILURE;
1629         else
1630             return SUCCESS;
1631     }
1632
1633     tmg_sweep(target_edge, other_edge, errorptr, calptr)
1634     int target_edge, other_edge, *errorptr;
1635     struct CALIB *calptr;
1636 {
1637     int thresh, current_errors, actual, count, min, max;
1638     current_errors = 0;
1639     min = tmg_predict_threshold(50000L,
1640         calptr->EdgeSlope[0][other_edge],
1641         calptr->EdgeOffset[0][other_edge]);
1642     max = tmg_predict_threshold(70000L,
1643         calptr->EdgeSlope[0][other_edge],
1644         calptr->EdgeOffset[0][other_edge]);
1645     for(thresh = min; thresh < max; thresh++)
1646     {
1647         tmgptr->edge_delay[other_edge].delay = thresh;
1648         for(count = 1000; count > 0; count--)
1649             if (tmg_detect(target_edge, 100L, &actual, DETECT_BOOL) != SUCCESS)
1650             {
1651                 current_errors++;
1652                 break;
1653             }
1654             if(actual == 0)
1655             {
1656                 current_errors++;
1657                 break;
1658             }
1659         }
1660     }
1661     if(current_errors)
1662     {
1663         *errorptr += current_errors;
1664         lm_message("delay line coupling detected between edges %d and %d\n",
1665             target_edge, other_edge);
1666     }
1667 }
1668

```

| | | | | |
|--|--|---------------------------------|--------------------|-----------------|
| Copyright 1989 Logic Modeling Systems | | HEADER FILE diags/tmg_extn.h | DATE 5/23/89 | PAGE # 1/191 |
| | | | TIME 4:41:35 pm | |
| LINE # | HEADER TEXT | | | |
| 1 | /* SOCS_ID: tmg_extn.h rev 3.1.1, 4/24/89 at 07:50:42 */ | | | |
| 2 | /* | | | |
| 3 | | | | |
| 4 | tmg_extn.h | | | |
| 5 | | | | |
| 6 | External declarations of global variables for the Timing | | | |
| 7 | Generator Diagnostics and Calibration | | | |
| 8 | Last Modified 7/18/88 -- wfk | | | |
| 9 | | | | |
| 10 | */ | | | |
| 11 | | | | |
| 12 | extern TMC *tmgptr; | | | |
| 13 | extern TMC_DIAG *tmgdiagptr; | | | |
| 14 | extern char mbf[]; | | | |
| 15 | extern int intflag; | | | |
| 16 | extern struct CALLS calib; | | | |
| 17 | extern u_char *tmg_ptr; | | | |
| 18 | extern u_char *tmg_kptr; | | | |
| 19 | extern u_char *tmg_arptr; | | | |
| 20 | extern u_long *tmg_periodptr; | | | |
| 21 | extern u_long tmg_max; | | | |
| 22 | extern int tmg_freq_table_built; | | | |
| 23 | extern u_long tmg_min_index1[]; | | | |
| 24 | extern u_long tmg_max_index1[]; | | | |
| 25 | extern u_long tmg_min_index2[]; | | | |
| 26 | extern u_long tmg_max_index2[]; | | | |
| 27 | extern u_long tmg_min_index3[]; | | | |
| 28 | extern u_long tmg_max_index3[]; | | | |
| 29 | extern u_long tmg_max_index4[]; | | | |
| 30 | | | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_glbl.c

DATE

5/23/89

PAGE #

TIME

4:41:35 pm

1/192

```

1  /* SCOS_ID: tmg_glbl.c rev 1.1, 4/24/89 at 07:50:50 */
2  /*
3  *.....
4  *
5  *      tmg_glbl.c
6  *
7  *      Global variables used in Timing Generator
8  *      Diagnostics and Calibration.
9  *
10 *.....
11 *
12 #include "common.h"
13 #include "tmg.h"
14 #include "tmg_def.h"
15
16 TMC *tmgptr = (TMC *)CLOCK_BASE; /* pointer to TMC */
17 TMC_DIAG *tmgdiagptr = (TMC_DIAG *)CLOCK_BASE; /* alternate pointer */
18
19 /* global variables used for generating/maintaining frequency table */
20 u_char *tmg_bp; /* pointer to k register values */
21 u_char *tmg_kptr; /* pointer to k register values */
22 u_char *tmg_sptr; /* pointer to clock select values */
23 u_long *tmg_periodptr; /* pointer to clock period values */
24 u_long *tmg_max; /* pointer to maximum index for above */
25 int tmg_freq_table_built = FALSE;
26 u_long *tmg_min_index[4]; /* range limits for searching during */
27 u_long *tmg_max_index[4]; /* calibration */
28 u_long *tmg_min_index2[4];
29 u_long *tmg_max_index2[4];
30 u_long *tmg_min_index3[4];
31 u_long *tmg_max_index3[4];
32
33 struct CALIB calib =
34 {
35     {
36         { 0, 0, 0, 0, 0, 0 },
37         { 0, 0, 0, 0, 0, 0 },
38         { 0, 0, 0, 0, 0, 0 },
39         { 0, 0, 0, 0, 0, 0 }
40     },
41     { 10000, 16000, 80000, 400000 }, /* EdgeMinThresh */
42     { 128000, 512000, 2560000, 12800000 }, /* EdgeMaxDelay */
43     {
44         { 0, 0, 0, 0, 0, 0 },
45         { 0, 0, 0, 0, 0, 0 },
46         { 0, 0, 0, 0, 0, 0 },
47         { 0, 0, 0, 0, 0, 0 }
48     },
49     /* EdgeOffset */
50     {
51         { 575, 575, 575, 575, 575, 575 },
52         { 2300, 2300, 2300, 2300, 2300, 2300 },
53         { 11500, 11500, 11500, 11500, 11500, 11500 },
54         { 57500, 57500, 57500, 57500, 57500, 57500 }
55     },
56     /* EdgeSlope */
57     {
58         { 0, 0, 0, 0, 0, 0 },
59         { 0, 0, 0, 0, 0, 0 },
60         { 0, 0, 0, 0, 0, 0 },
61         { 0, 0, 0, 0, 0, 0 }
62     },
63     /* EdgeLinearity */
64     { 575, 2300, 11500, 57500 },
65     /* Edge7MinThresh */
66     { 0, 0, 0, 0 },
67     /* Edge7Slope */
68     { 0, 0, 0, 0 },
69     /* Edge7Linearity */
70     { 0, 0, 0, 0 },
71     /* SampleMinThresh */
72     { 150000, 62000, 126000, 446000 },
73     /* SampleMinDelay */
74     { 128000, 256000, 512000, 1024000 },
75     /* SampleMaxDelay */
76     { 500, 1000, 2000, 4000 },
77     /* SampleSlope */
78     { 0, 0, 0, 0 },
79     /* SampleLinearity */
80     {
81         { 30000, 30000, 30000, 30000 },
82         { 30000, 30000, 30000, 30000 },
83         { 30000, 30000, 30000, 30000 },
84         { 30000, 30000, 30000, 30000 }
85     },
86     /* SampleOffset */
87     {
88         { 9000, 13000, 21000, 17000 },
89         { 110000, 220000, 440000, 880000 },
90         { 0, 0, 0, 0 }
91     },
92     /* EarlySampleMinDelay */
93     /* EarlySampleMaxDelay */
94     /* EarlySampleOffset */
95     /* DelayDelay */
96     /* DumpDelay */
97     /* CalCompleted */
98 };
99
100 struct CALIB default_calib =
101 {
102     {
103         { 0, 0, 0, 0, 0, 0 },
104         { 0, 0, 0, 0, 0, 0 },
105         { 0, 0, 0, 0, 0, 0 },
106         { 0, 0, 0, 0, 0, 0 }
107     },
108     /* EdgeMinThresh */
109     { 10000, 16000, 80000, 400000 },
110     /* EdgeMinDelay */
111     { 128000, 512000, 2560000, 12800000 },
112     /* EdgeMaxDelay */
113     {
114         { 0, 0, 0, 0, 0, 0 },
115         { 0, 0, 0, 0, 0, 0 },
116         { 0, 0, 0, 0, 0, 0 },
117         { 0, 0, 0, 0, 0, 0 }
118     },
119     /* EdgeOffset */
120     {
121         { 575, 575, 575, 575, 575, 575 },
122         { 2300, 2300, 2300, 2300, 2300, 2300 },
123         { 11500, 11500, 11500, 11500, 11500, 11500 },
124         { 57500, 57500, 57500, 57500, 57500, 57500 }
125     },
126     /* EdgeSlope */
127     {
128         { 0, 0, 0, 0, 0, 0 },
129         { 0, 0, 0, 0, 0, 0 },
130         { 0, 0, 0, 0, 0, 0 },
131         { 0, 0, 0, 0, 0, 0 }
132     },
133     /* EdgeLinearity */
134     { 575, 2300, 11500, 57500 },
135     /* Edge7MinThresh */
136     { 0, 0, 0, 0 },
137     /* Edge7Slope */
138     { 0, 0, 0, 0 },
139     /* Edge7Linearity */
140     { 0, 0, 0, 0 },
141     /* SampleMinThresh */
142     { 150000, 62000, 126000, 446000 },
143     /* SampleMinDelay */
144     { 128000, 256000, 512000, 1024000 },
145     /* SampleMaxDelay */
146     { 500, 1000, 2000, 4000 },
147     /* SampleSlope */
148     { 0, 0, 0, 0 },
149     /* SampleLinearity */

```

| | | | | |
|--|-----------------------------------|-------------------------------------|-----------------|-----------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/trng_glbl.c | DATE 5/23/89 | PAGE # 2/193 |
| TIME 4:41:35 pm | | | | |
| LINE # | SOURCE TEXT | | | |
| 121 | { | | | |
| 122 | {30000, 30000, 30000, 30000}, | | | |
| 123 | {30000, 30000, 30000, 30000}, | | | |
| 124 | {30000, 30000, 30000, 30000}, | | | |
| 125 | {30000, 30000, 30000, 30000} | | | |
| 126 | /* SampleOffset */ | | | |
| 127 | #ifdef EARLYSAMPLETRIGGER | | | |
| 128 | {9000, 13000, 21000, 37000}, | | | |
| 129 | {110000, 220000, 440000, 880000}, | | | |
| 130 | {0, 0, 0, 0}, | | | |
| 131 | #endif | | | |
| 132 | 0x22, | | | |
| 133 | 7, | | | |
| 134 | 0 | | | |
| 135 | /* DelayDelay */ | | | |
| | /* DumpDelay */ | | | |
| | /* CalCompleted */ | | | |
| | /* | | | |
| | */ | | | |
| | { | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_menu.c

DATE 5/23/89 PAGE #
TIME 4:41:35 pm 1/194

LINE # SOURCE TEXT

1 /* SCCS ID: tmg_menu.c rev 3.2, 5/9/89 at 15:55:01 */

2 /*.....*/

3 /*.....*/

4 /*.....*/

5 /*.....*/

6 /*.....*/

7 /*.....*/

8 /*.....*/

9 /*.....*/

10 /*.....*/

11 #include <math.h>

12 #include "common.h"

13 #include "tmg.h"

14 #include "tmg_def.h"

15 #include "vrtx.h"

16 #include "lm_diags.h"

17 #include "tmg_exts.h"

18

19 tmg_diag_disp(parent_menu)

20 LM_DIAG_MENU "parent_menu,

21 {

22 extern int tmg_register_test(),

23 tmg_idprom_test(),

24 tmg_clock_sync_test(),

25 tmg_ctc_test(),

26 tmg_pll_rate(),

27 tmg_pll_div(),

28 tmg_pll_lock(),

29 tmg_slow_detect(),

30 tmg_interrupts(),

31 tmg_calibrate(),

32 tmg_diag_disp2(),

33 tmg_delay_line_isolation();

34

35 static LM_DIAG_MENU_ITEM menu_list[] =

36 {

37 {

38 "1",

39 "Register Test",

40 tmg_register_test,

41 LM_DIAG_diag_routine,

42 LM_DIAG_null

43 },

44 {

45 "2",

46 "ID Prom Test",

47 tmg_idprom_test,

48 LM_DIAG_diag_routine,

49 LM_DIAG_null

50 },

51 {

52 "3",

53 "Clock Synchronizer Test",

54 tmg_clock_sync_test,

55 LM_DIAG_diag_routine,

56 LM_DIAG_null

57 },

58 {

59 "4",

60 "Counter Timer Test",

61 tmg_ctc_test,

62 LM_DIAG_diag_routine,

63 LM_DIAG_null

64 },

65 {

66 "5",

67 "Phase Locked Loop Rate",

68 tmg_pll_rate,

69 LM_DIAG_diag_routine,

70 LM_DIAG_null

71 },

72 {

73 "6",

74 "Phase Locked Loop Division",

75 tmg_pll_div,

76 LM_DIAG_diag_routine,

77 LM_DIAG_null

78 },

79 {

80 "7",

81 "Phase Locked Loop Lock Time",

82 tmg_pll_lock,

83 LM_DIAG_diag_routine,

84 LM_DIAG_null

85 },

86 {

87 "8",

88 "Slow Clock Detector Test",

89 tmg_slow_detect,

90 LM_DIAG_diag_routine,

91 LM_DIAG_null

92 },

93 {

94 "9",

95 "Check Interrupt Sources",

96 tmg_interrupts,

97 LM_DIAG_diag_routine,

98 LM_DIAG_null

99 },

100 {

101 "10",

102 "Check Calibration",

103 tmg_calibrate,

104 LM_DIAG_diag_routine,

105 LM_DIAG_null

106 },

107 {

108 "11",

109 "Check Edge Isolation",

110 tmg_delay_line_isolation,

111 LM_DIAG_diag_routine,

112 LM_DIAG_null

113 },

114 {

115 "12",

116 "Utilities",

117 tmg_diag_disp2,

118 LM_DIAG_utility_menu,

119 LM_DIAG_null

120 }

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_menu.c

*

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:35 pm | 2/195 |

| LINE # | SOURCE TEXT |
|--------|-------------|
|--------|-------------|

```

121  },
122  static LM_DIAG_MENU menu =
123  {
124      "TIMING GENERATOR DIAGNOSTICS",
125      sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
126      0,
127      menu_list
128  },
129  },
130  menu.title = parent_menu->
131  menu.items[parent_menu->current_selection].menu_text,
132  return lm_display_menu(&menu);
133  }
134  }
135  }
136  tmg_diag_disp2(parent_menu)
137  LM_DIAG_MENU *parent_menu,
138  {
139      int tmg_freq_ext0();
140      int tmg_freq_ext1();
141      int pac_sel_pat_clk();
142      int tmg_select_mode();
143      int tmg_select_ramps();
144      int tmg_select_thresholds();
145      int tmg_select_slot_count();
146      int tmg_loop();
147      int print_calib_structure();
148      int tmg_plot_delays();
149      int tmg_modify_debug_flag();
150      int tmg_jitter_test();
151      int tmg_scope_measurements();
152  }
153  static LM_DIAG_MENU_ITEM menu_list[] =
154  {
155      {
156          "1",
157          "Measure Ext0 Frequency",
158          tmg_freq_ext0,
159          LM_DIAG_utility,
160          LM_DIAG_null
161      },
162      {
163          "2",
164          "Measure Ext1 Frequency",
165          tmg_freq_ext1,
166          LM_DIAG_utility,
167          LM_DIAG_null
168      },
169      {
170          "3",
171          "Select Pattern Clock Period",
172          pac_sel_pat_clk,
173          LM_DIAG_utility,
174          LM_DIAG_null
175      },
176      {
177          "4",
178          "Select Test/Sample Modes",
179          tmg_select_mode,
180          LM_DIAG_utility,
181          LM_DIAG_null
182      },
183      {
184          "5",
185          "Select Edge/Sample Ramps",
186          tmg_select_ramps,
187          LM_DIAG_utility,
188          LM_DIAG_null
189      },
190      {
191          "6",
192          "Select Edge/Sample Thresholds",
193          tmg_select_thresholds,
194          LM_DIAG_utility,
195          LM_DIAG_null
196      },
197      {
198          "7",
199          "Select Slot Count",
200          tmg_select_slot_count,
201          LM_DIAG_utility,
202          LM_DIAG_null
203      },
204      {
205          "8",
206          "PLAY",
207          tmg_loop,
208          LM_DIAG_utility & "LM_DIAG_no_repeat",
209          LM_DIAG_null
210      },
211      {
212          "9",
213          "Print Current Calibration Structure",
214          print_calib_structure,
215          LM_DIAG_utility,
216          LM_DIAG_null
217      },
218      {
219          "10",
220          "Plot Delay Line Delay vs Clock Frequency",
221          tmg_plot_delays,
222          LM_DIAG_utility,
223          LM_DIAG_null
224      },
225      {
226          "11",
227          "Enable/Disable Optional Debugging Messages Menu",
228          tmg_modify_debug_flag,
229          LM_DIAG_utility_menu,
230          LM_DIAG_null
231      },
232      {
233          "12",
234          "Jitter Measurement",
235          tmg_jitter_test,
236          LM_DIAG_utility,
237          LM_DIAG_null
238      },
239      {
240          "13",

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_menu.c | DATE 5/23/89 TIME 4:41:35 pm | PAGE # 3/196 |
|--|---|------------------------------------|---------------------------------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 241 | "Oscilloscope Measurements Menu", | | | |
| 242 | tmg_scope_measurements, | | | |
| 243 | LM_DIAG_utility_menu, | | | |
| 244 | LM_DIAG_null | | | |
| 245 | } | | | |
| 246 | }, | | | |
| 247 | static LM_DIAG_MENU menu = | | | |
| 248 | { | | | |
| 249 | "TIMING GENERATOR UTILITIES", | | | |
| 250 | sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM), | | | |
| 251 | 0, | | | |
| 252 | menu_list | | | |
| 253 | }, | | | |
| 254 | menu.title = parent_menu-> | | | |
| 255 | menu.items[parent_menu->current_selection].menu_text, | | | |
| 256 | return lm_display_menu(&menu); | | | |
| 257 | } | | | |
| 258 | return lm_display_menu(&menu); | | | |
| 259 | } | | | |
| 260 | tmg_modify_debug_flag(parent_menu) | | | |
| 261 | LM_DIAG_MENU *parent_menu, | | | |
| 262 | { | | | |
| 263 | extern int tmg_modify_edge_related_debug_flag(), | | | |
| 264 | tmg_modify_sample_related_debug_flag(), | | | |
| 265 | tmg_modify_delayline_related_debug_flag(), | | | |
| 266 | tmg_modify_misc_related_debug_flag(), | | | |
| 267 | tmg_modify_all_debug_flag(), | | | |
| 268 | static LM_DIAG_MENU_ITEM menu_list[] = | | | |
| 269 | { | | | |
| 270 | { | | | |
| 271 | "1", | | | |
| 272 | "Edge Calibration Related Debug Flags", | | | |
| 273 | tmg_modify_edge_related_debug_flag, | | | |
| 274 | LM_DIAG_utility LM_DIAG_automatic_quit, | | | |
| 275 | LM_DIAG_null | | | |
| 276 | }, | | | |
| 277 | { | | | |
| 278 | "2", | | | |
| 279 | "Sample Calibration Related Debug Flags", | | | |
| 280 | tmg_modify_sample_related_debug_flag, | | | |
| 281 | LM_DIAG_utility LM_DIAG_automatic_quit, | | | |
| 282 | LM_DIAG_null | | | |
| 283 | }, | | | |
| 284 | { | | | |
| 285 | "3", | | | |
| 286 | "Delay Line Calibration Related Debug Flags", | | | |
| 287 | tmg_modify_delayline_related_debug_flag, | | | |
| 288 | LM_DIAG_utility LM_DIAG_automatic_quit, | | | |
| 289 | LM_DIAG_null | | | |
| 290 | }, | | | |
| 291 | { | | | |
| 292 | "4", | | | |
| 293 | "Miscellaneous Calibration Related Debug Flags", | | | |
| 294 | tmg_modify_misc_related_debug_flag, | | | |
| 295 | LM_DIAG_utility LM_DIAG_automatic_quit, | | | |
| 296 | LM_DIAG_null | | | |
| 297 | }, | | | |
| 298 | { | | | |
| 299 | "5", | | | |
| 300 | "Set/Reset All Debug Flags", | | | |
| 301 | tmg_modify_all_debug_flag, | | | |
| 302 | LM_DIAG_utility LM_DIAG_automatic_quit, | | | |
| 303 | LM_DIAG_null | | | |
| 304 | }, | | | |
| 305 | } | | | |
| 306 | }, | | | |
| 307 | static LM_DIAG_MENU menu = | | | |
| 308 | { | | | |
| 309 | "TIMING GENERATOR DEBUG FLAGS", | | | |
| 310 | sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM), | | | |
| 311 | 0, | | | |
| 312 | menu_list | | | |
| 313 | }, | | | |
| 314 | menu.title = parent_menu-> | | | |
| 315 | menu.items[parent_menu->current_selection].menu_text, | | | |
| 316 | return(lm_display_menu(&menu)); | | | |
| 317 | } | | | |
| 318 | tmg_scope_measurements(parent_menu) | | | |
| 319 | LM_DIAG_MENU *parent_menu, | | | |
| 320 | { | | | |
| 321 | int tmg_continuousclocktest(), | | | |
| 322 | int tmg_dac_settle(), | | | |
| 323 | int tmg_edge_jitter(), | | | |
| 324 | int tmg_sample_jitter(), | | | |
| 325 | int tmg_edge_alignment(), | | | |
| 326 | static LM_DIAG_MENU_ITEM menu_list[] = | | | |
| 327 | { | | | |
| 328 | { | | | |
| 329 | "1", | | | |
| 330 | "Continuous Phase Lock Loop Lock/Unlock", | | | |
| 331 | tmg_continuousclocktest, | | | |
| 332 | LM_DIAG_utility, | | | |
| 333 | LM_DIAG_null | | | |
| 334 | }, | | | |
| 335 | { | | | |
| 336 | "2", | | | |
| 337 | "D/A Converter Settling Time", | | | |
| 338 | tmg_dac_settle, | | | |
| 339 | LM_DIAG_utility, | | | |
| 340 | LM_DIAG_null | | | |
| 341 | }, | | | |
| 342 | { | | | |
| 343 | "3", | | | |
| 344 | "Edge Jitter Measurement", | | | |
| 345 | tmg_edge_jitter, | | | |
| 346 | LM_DIAG_utility, | | | |
| 347 | LM_DIAG_null | | | |
| 348 | }, | | | |
| 349 | { | | | |
| 350 | "4", | | | |
| 351 | "Sample Jitter Measurement", | | | |
| 352 | tmg_sample_jitter, | | | |
| 353 | LM_DIAG_utility, | | | |
| 354 | LM_DIAG_null | | | |
| 355 | }, | | | |
| 356 | { | | | |
| 357 | "5", | | | |
| 358 | "Set/Reset All Debug Flags", | | | |
| 359 | tmg_modify_all_debug_flag, | | | |
| 360 | LM_DIAG_utility, | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_menu.c

DATE 5/23/89
TIME 4:41:35 pm

PAGE #
4/197

| LINE # | SOURCE TEXT |
|--------|---|
| 361 | { |
| 362 | "5", |
| 363 | "Edge Time Alignment", |
| 364 | tmg_edge_alignment, |
| 365 | LM_DIAG_utility, |
| 366 | LM_DIAG_null |
| 367 | } |
| 368 | }, |
| 369 | |
| 370 | static LM_DIAG_MENU menu = |
| 371 | { |
| 372 | "TIMING GENERATOR OSCILLOSCOPE MEASUREMENTS", |
| 373 | sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM), |
| 374 | 0, |
| 375 | menu_list |
| 376 | }, |
| 377 | menu.title = parent_menu-> |
| 378 | menu_items[parent_menu->current_selection].menu_text, |
| 379 | |
| 380 | return lm_display_menu(&menu); |
| 381 | } |
| 382 | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_run.c

DATE

5/23/89

PAGE #

TIME

4:41:35 pm

1/198

```

1  /* SOCS_ID: tmg_run.c rev 3.1.1, 4/24/89 at 07:50:56 */
2  /*
3  * .....
4  *      tmg_run.c
5  *      Timing Generator Run-time support routines
6  *      .....
7  * .....
8  * .....
9  * .....
10 #include "common.h"
11 #include "tmg.h"
12 #include "message.h"
13 #include "tmg_def.h"
14 #include "tmg_run.h"
15 #include "tmg_extn.h"
16 #include "vtr.h"
17 #include "math.h"
18
19 extern u_char play_completed_flag;
20 u_char post_end_of_play;
21 extern int play_semaphore;
22 static int zero = 0;
23
24 #define DEBUG
25
26 #ifdef DEBUG
27 #define DPRINTF(x) printf x
28 #else
29 #define DPRINTF(x) /* do nothing */
30 #endif
31
32
33 /*
34 *      In tmg_get_frequency_setting()
35 *
36 *      This routine computes the values for n, k, and clock
37 *      select register setting to give the best approximation
38 *      of the desired clock period. The clock period provided
39 *      is always equal to or greater than that requested.
40 *      The actual clock period is related to n, k as follows:
41 *      period = (k / n) * T_REF, where
42 *      k = 1, 2, ..., 256 or k = 2, 4, ..., 512,
43 *      n = 128, 129, ..., 256, and
44 *      T_REF = 256 / 30 MHz = PLL reference period.
45 *
46 *      Returns: FAILURE if period out of range, else SUCCESS
47 */
48
49 In tmg_get_frequency_setting(period, actualptr, jitterptr, sptr, kptr, selectptr)
50 u_long period; /* physical clock period in ps */
51 u_long *actualptr; /* actual clock period in ps */
52 u_long *jitterptr; /* jitter per period in ps */
53 u_long *sptr; /* select register */
54 u_long *kptr;
55 u_long *selectptr;
56 {
57     double error, besterror, kprime;
58     int tmpk, n, save_k, save_n;
59
60     if((period > MAX_PERIOD) || (period < MIN_PERIOD)) {
61         In_queue_message(ERROR_MSG, "period out of range, period = %d",
62             period);
63         return(FAILURE);
64     }
65
66     for(besterror = 1.0, n = N_MIN, k = N_MAX, n++)
67     {
68         kprime = n * period / T_REF;
69         if((tmpk = (int)ceil(kprime)) > 512) /* see if out of range */
70             continue; /* if so, ignore it */
71         if(tmpk > 256) /* see if above 256 */
72             if(tmpk & 1) /* see if odd */
73                 tmpk++; /* make even */
74         if(error = (((double)tmpk - kprime) / kprime) < besterror)
75         {
76             save_n = n;
77             save_k = tmpk;
78             besterror = error;
79         }
80     }
81     *sptr = 256 - save_n + 1; /* value to put in reg */
82     if(save_k == 1)
83     {
84         *kptr = 255; /* special case */
85         *selectptr = 0; /* select div by 2 */
86     }
87     else
88     {
89         if(save_k > 256)
90         {
91             *kptr = 256 - (save_k >> 1) + 1; /* divide k by 2 */
92             *selectptr = 2; /* select div by 4k */
93         }
94         else
95         {
96             *kptr = 256 - save_k + 1; /* normal case */
97             *selectptr = 1; /* select div by 2k */
98         }
99     }
100     *actualptr = T_REF * save_k / save_n;
101     *jitterptr = 25 * 2 * save_k; /* 20 ps per PLL clock */
102     return(SUCCESS);
103 }
104
105 /*
106 *      In tmg_get_sample_ramp_delay()
107 *
108 *      This function returns delay of SAMPLE with the given threshold.
109 *      This contains an offset error and should not be used to compute
110 *      absolute delays. Using this function to compute a difference
111 *      in time is OK since the offset errors cancel. Delays are in ps.
112 *
113 *      Inputs: range (1 of 4 ramp rates), threshold value, pointer to
114 *      delay variable, pointer to error
115 *      Outputs: delay and error are assigned, function returns
116 *      SUCCESS or FAILURE
117 */
118 In tmg_get_sample_ramp_delay(rate, threshold, delayptr, errorptr)
119 u_char rate;
120 u_char threshold;
121 long *delayptr, *errorptr;
122

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_run.c | DATE 5/23/89 | PAGE # 2/199 |
|--|--|--|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 121 | | if((rate > 3) (threshold < calib.sampleMinThresh(rate))) { | | |
| 122 | | in_queue_message(ERROR_MSG, "rate too large OR threshold too small"); | | |
| 123 | | return(FAILURE); | | |
| 124 | | } | | |
| 125 | | errorptr = 0; /* tentatively not used */ | | |
| 126 | | if(tmgptr->sample_mode) | | |
| 127 | | { | | |
| 128 | | *delayptr = threshold + calib.sampleSlope(rate); | | |
| 129 | | /* errorptr = calib.sampleSlope(rate); */ | | |
| 130 | | } | | |
| 131 | | else | | |
| 132 | | { | | |
| 133 | | *delayptr = threshold + calib.sampleSlope(rate) | | |
| 134 | | + calib.earlySampleOffset(rate); | | |
| 135 | | /* errorptr = calib.earlySampleSlope(rate) + 750; */ | | |
| 136 | | } | | |
| 137 | | return(SUCCESS); | | |
| 138 | | } | | |
| 139 | | /* | | |
| 140 | | in_tmg_get_early_sample_thres() | | |
| 141 | | /* | | |
| 142 | | This function returns threshold of early SAMPLE gives the delay. | | |
| 143 | | Inputs: ramp (1 of 4 ramp rates), delay value, pointer to | | |
| 144 | | threshold variable | | |
| 145 | | Outputs: threshold is assigned, function returns | | |
| 146 | | SUCCESS or FAILURE | | |
| 147 | | */ | | |
| 148 | | in_tmg_get_early_sample_thres(rate, delay, thresholdptr) | | |
| 149 | | u_char rate; | | |
| 150 | | u_long delay; | | |
| 151 | | u_char *thresholdptr; | | |
| 152 | | { | | |
| 153 | | u_long max_delay; | | |
| 154 | | max_delay = 255 + calib.sampleSlope(rate) + | | |
| 155 | | calib.earlySampleOffset(rate); | | |
| 156 | | if((rate > 3) (delay > max_delay)) { | | |
| 157 | | in_queue_message(ERROR_MSG, "rate too large OR delay too large"); | | |
| 158 | | return(FAILURE); | | |
| 159 | | } | | |
| 160 | | *thresholdptr = (delay - calib.earlySampleOffset(rate)) / | | |
| 161 | | calib.sampleSlope(rate); | | |
| 162 | | return(SUCCESS); | | |
| 163 | | } | | |
| 164 | | /* | | |
| 165 | | in_tmg_get_dead_time() | | |
| 166 | | /* | | |
| 167 | | This function returns dead time in ps for specified clock period | | |
| 168 | | in ps. Currently, this function returns dead time that depends | | |
| 169 | | only upon the EDGE 0 slope of ramp needed to open the clock period. | | |
| 170 | | Start dead time is due to non-zero minimum thresholds for the | | |
| 171 | | ladder ramp settings. Ending dead time is due to finite dump time | | |
| 172 | | of the ramp. | | |
| 173 | | Inputs: period in ps, pointers to start and ending dead times | | |
| 174 | | Outputs: dead times are assigned, function returns SUCCESS or FAILURE | | |
| 175 | | */ | | |
| 176 | | in_tmg_get_dead_time(period, startdeadptr, enddeadptr) | | |
| 177 | | u_long period; | | |
| 178 | | u_long *startdeadptr; | | |
| 179 | | u_long *enddeadptr; | | |
| 180 | | { | | |
| 181 | | register u_long ramp; | | |
| 182 | | if((period < (u_long)MIN_PERIOD) (period > (u_long)MAX_PERIOD)) { | | |
| 183 | | in_queue_message(ERROR_MSG, "period out of range; period = %d", | | |
| 184 | | period); | | |
| 185 | | return(FAILURE); | | |
| 186 | | } | | |
| 187 | | for(ramp = 0; ramp < 4; ramp++) | | |
| 188 | | { | | |
| 189 | | if(calib.EdgeMaxDelay[ramp] > period) | | |
| 190 | | { | | |
| 191 | | in_tmg_get_ramp_dead_time(period, ramp, startdeadptr, enddeadptr); | | |
| 192 | | return(SUCCESS); | | |
| 193 | | } | | |
| 194 | | } | | |
| 195 | | in_queue_message(ERROR_MSG, "period not reached; period = %d, EdgeMaxDelay[3] = %d", | | |
| 196 | | period, calib.EdgeMaxDelay[3]); | | |
| 197 | | return(FAILURE); | | |
| 198 | | } | | |
| 199 | | /* | | |
| 200 | | in_tmg_get_ramp_dead_time(period, ramp, startdeadptr, enddeadptr) | | |
| 201 | | register u_long period, ramp; | | |
| 202 | | u_long *startdeadptr; | | |
| 203 | | u_long *enddeadptr; | | |
| 204 | | { | | |
| 205 | | int i; | | |
| 206 | | long min_offset; | | |
| 207 | | /* start dead time = edge minimum delay | | |
| 208 | | *startdeadptr = calib.EdgeMinDelay[ramp]; | | |
| 209 | | /* end dead time = ramp dump time | | |
| 210 | | /* + ramp uncertainty | | |
| 211 | | /* + 2 * edge jitter | | |
| 212 | | /* - min of all offsets of ramp 0 | | |
| 213 | | /* Note: this should also include clock jitter, but | | |
| 214 | | /* this routine is unaware of clock frequency. | | |
| 215 | | /* This is accounted for elsewhere in runtime code | | |
| 216 | | min_offset = calib.EdgeOffset[0][0]; | | |
| 217 | | for(i = 1; i < NUMBER_OF_EDGES; i++) | | |
| 218 | | { | | |
| 219 | | if(min_offset > calib.EdgeOffset[0][i]) | | |
| 220 | | min_offset = calib.EdgeOffset[0][i]; | | |
| 221 | | } | | |
| 222 | | *enddeadptr = (long)RAMP_DUMP_TIME | | |
| 223 | | + ((long)period / 50); | | |
| 224 | | + 2 * calib.EdgeSlope[ramp][0] /* + 2 thresholds | | |
| 225 | | + 2 * calib.EdgeSlope[ramp][0] /* + 2 * edge jitter | | |
| 226 | | - min_offset; | | |
| 227 | | return(SUCCESS); | | |
| 228 | | } | | |
| 229 | | /* | | |
| 230 | | in_tmg_get_falling_sample_setting() | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_run.c

DATE

5/23/89

PAGE #

TIME

4:41:35 pm

3/200

```

LINE #          SOURCE TEXT
241 *
242 * This function returns the value to load into Timing Generator
243 * Register 7. The delay amount is equal to or greater than that
244 * requested. Software must guarantee that this delay is greater
245 * than the delay for the leading edge of SAMPLE.
246 *
247 * Inputs: period of clock in ps, delay requested in ps (referenced
248 * from last pattern), pointer to width setting
249 * Outputs: width is assigned, if delay is possible, function returns
250 * SUCCESS, else FAILURE
251 */
252 int tmg_get_falling_sample_setting(period, delay, widthptr)
253 u_long period;
254 u_long delay;
255 u_long *widthptr;
256 {
257     unsigned int tmp;
258     if((tmp = (unsigned int)((double)delay / (double)period)) > 255) {
259         /* requested delay of 64 ps requires 64 (255) clocks,
260          * delay, tmp;
261         */
262         return(FAILURE);
263     }
264     else
265     {
266         *widthptr = tmp;
267         return(SUCCESS);
268     }
269 }
270
271 /*
272  * tmg_get_edge_setting()
273  *
274  * This function computes edge settings and selects appropriate
275  * ramp (edge 0) slope. All delays are less than or equal to
276  * what is requested.
277  *
278  * Inputs: logical clock period in ps, Array of desired edge
279  *         delay times in ps, pointer to array of threshold
280  *         settings to generate desired delays
281  *
282  * Outputs: threshold array is assigned along with quantization
283  *          and extra delay, function returns SUCCESS or FAILURE
284  */
285 int tmg_get_edge_setting(period, delay, threshold)
286 u_long period;
287 u_long delay[]; /* better be size 5 */
288 u_long threshold[];
289 {
290     int i, j;
291     u_long start, end, thresh;
292     if(!tmg_get_dead_time(period, &start, &end) != SUCCESS) {
293         return(FAILURE);
294     }
295     for(i = 0; i < 4; i++)
296     {
297         if(calib.EdgeMaxDelay[i] > period)
298             break;
299     }
300     if(i == 4) {
301         /* period too large, period = 64, EdgeMaxDelay[3] = 64,
302          * period, calib.EdgeMaxDelay[3];
303         */
304         return(FAILURE);
305     }
306     for(j = 0; j < 6; j++)
307     {
308         if((delay[j] < calib.EdgeMinDelay[i]) || (delay[j] > (period - end))) {
309             /* delay out of range, delay[6] = 64, EdgeMinDelay = 64, max = 64,
310              * delay[j], calib.EdgeMinDelay[i], (period - end);
311             */
312             return(FAILURE);
313         }
314         threshold[j] = (delay[j] - calib.EdgeOffset[i][j]) / calib.EdgeSlope[i][0];
315     }
316     return(SUCCESS);
317 }
318
319 /*
320  * tmg_get_sample_setting()
321  *
322  * This function computes edge 7 settings.
323  *
324  * Inputs: logical clock period in ps, desired sample delay
325  *         time in ps, pointer to threshold setting to generate
326  *         desired delay.
327  *
328  * Outputs: threshold is assigned, function returns SUCCESS or FAILURE
329  */
330 int tmg_get_sample_setting(period, delay, thresholdptr)
331 u_long period;
332 u_long delay;
333 u_long *thresholdptr;
334 {
335     int eramp, edge, aramp;
336     u_long start, end;
337     long thresh;
338     long max_thresh, tmp;
339     if(!tmg_get_dead_time(period, &start, &end) != SUCCESS)
340         return(FAILURE);
341     for(eramp = 0; eramp < 4; eramp++)
342     {
343         if(calib.EdgeMaxDelay[eramp] > period)
344             break;
345     }
346     if(eramp == 4) {
347         /* period too large, period = 64, EdgeMaxDelay[3] = 64,
348          * period, calib.EdgeMaxDelay[3];
349         */
350         return(FAILURE);
351     }
352     /* find maximum of all edge thresholds
353     * use this to bound allowable values for Edge7 threshold
354     */
355     for(max_thresh = 0; edge = 0; edge < NUMBER_OF_EDGES; edge++)
356     {
357         tmp = ((long)period - (long)end - calib.EdgeOffset[eramp][edge]);
358     }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_run.c

DATE 5/23/89
TIME 4:41:35 pm

PAGE #
4/201

```

LINE #          SOURCE TEXT
361      / calib.Edgeslope[eramp][edge],
362      if(max_thresh < tmp)
363      max_thresh = tmp;
364      }
365      if(delay < calib.SampleMinDelay[eramp])
366      {
367          ln_queue_message(ERROR_MSG, "delay out of range, delay[6] = %d, SampleMinDelay = %d, max = %d",
368          delay, calib.SampleMinDelay[eramp], (period - end));
369          return(FAILURE);
370      }
371      }
372      else
373      {
374          *thresholdptr = 355;
375          for(eramp = 0; eramp < 4; eramp++)
376          {
377              thresh = ((long)delay - calib.SampleOffset[eramp][eramp]
378              - calib.SampleSlope[eramp]*calib.SampleMinThresh[eramp])
379              / calib.Edge7Slope[eramp];
380              if((thresh < *thresholdptr) && (thresh > 0))
381                  *thresholdptr = thresh;
382          }
383          if(*thresholdptr < calib.Edge7MinThresh[0])
384          {
385              ln_queue_message(ERROR_MSG, "requested Edge 7 threshold below minimum");
386              return FAILURE;
387          }
388          if(*thresholdptr > max_thresh)
389          {
390              ln_queue_message(ERROR_MSG, "requested Edge 7 threshold %d is above \
391              maximum for period %d", *thresholdptr, period);
392              return FAILURE;
393          }
394      }
395      return SUCCESS;
396      }
397      }
398      }
399      }
400      /* ln_tmg_get_quantization_and_jitter_error(period, quantptr, jitterptr)
401      *
402      * This function returns the quantization step size of delay
403      * with respect to threshold setting, everything in ps.
404      *
405      * Inputs: logical clock period, pointer to error
406      * Outputs: quantization and jitter are assigned based on the edge 0
407      * slope of some ramp, returns SUCCESS or FAILURE
408      */
409      ln_tmg_get_quantization_and_jitter_error(period, quantptr, jitterptr)
410      u_long period;
411      u_long *quantptr;
412      u_long *jitterptr;
413      {
414          int i;
415          for(i = 0; i < 4; i++)
416          {
417              if(calib.EdgeMaxDelay[i] > period)
418                  break;
419          }
420          if(i == 4) {
421              ln_queue_message(ERROR_MSG, "period too large, period = %d, EdgeMaxDelay[3] = %d",
422              period, calib.EdgeMaxDelay[3]);
423              return(FAILURE);
424          }
425          *quantptr = calib.Edgeslope[i][0]; /* always 1 threshold worth */
426          *jitterptr = calib.Edgeslope[i][0]; /* 1 threshold for now */
427          return(SUCCESS);
428      }
429      }
430      }
431      }
432      /* ln_tmg_get_minimum_sample_ramp_setting()
433      *
434      * This function returns minimum threshold settings for the
435      * four sample ramps.
436      *
437      * Inputs: pointers to minimum threshold variables
438      * Outputs: threshold variables are assigned, returns SUCCESS
439      * or FAILURE
440      */
441      ln_tmg_get_minimum_sample_ramp_setting(r0ptr, r1ptr, r2ptr, r3ptr)
442      u_char *r0ptr;
443      u_char *r1ptr;
444      u_char *r2ptr;
445      u_char *r3ptr;
446      {
447          if(!calib.CalCompleted) {
448              ln_queue_message(ERROR_MSG, "problem was encountered during calibration");
449              return(FAILURE);
450          }
451          *r0ptr = calib.SampleMinThresh[0];
452          *r1ptr = calib.SampleMinThresh[1];
453          *r2ptr = calib.SampleMinThresh[2];
454          *r3ptr = calib.SampleMinThresh[3];
455          return(SUCCESS);
456      }
457      }
458      }
459      /* ln_tmg_get_sample_ramp_dead_time()
460      *
461      * This function returns the maximum dead time in ps for the SAMPLE
462      * ramps.
463      *
464      * Inputs: logical clock period, mode, pointer to dead time variable
465      * Outputs: dead time assigned, function returns SUCCESS or FAILURE
466      */
467      ln_tmg_get_sample_ramp_dead_time(period, mode, deadptr, jitterptr)
468      u_long period;
469      u_long mode;
470      u_long *deadptr;
471      u_long *jitterptr;
472      {
473          int eramp;
474          for(eramp = 0; eramp < 4; eramp++)
475          {
476              if(calib.EdgeMaxDelay[eramp] > period)
477                  break;
478          }
479          if(eramp == 4) {
480              ln_queue_message(ERROR_MSG, "period too large, period = %d, EdgeMaxDelay[3] = %d",
481              period, calib.EdgeMaxDelay[3]);
482              return(FAILURE);
483          }
484      }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_run.c

DATE

5/23/89

PAGE #

TIME 4:41:35 pm

5/202

```

LINE # SOURCE TEXT
481 )
482
483 if(!calib.CalCompleted) {
484     lm_queue_message(ERROR_MSG, "problem was encountered during calibration");
485     return(FAILURE);
486 }
487 if(mode == EDGE7SAMPLETRIGGERMODE)
488 {
489     if(eramp == 0) /* don't need to include delay line effects */
490         *deadptr = calib.SampleMinDelay(eramp);
491     else
492         *deadptr = calib.SampleMinDelay(eramp) + 1500L; /* add 1 + 3/4 ns */
493 }
494 else
495 {
496     if(eramp == 0)
497         *deadptr = 0; /* ok to place edges at magic chip dead time */
498     else
499         *deadptr = 2 * (calib.SampleMinDelay(eramp) + 1500L);
500     /* make sure this is greater than EDGE7 mode dead time */
501 }
502
503 *jitterptr = calib.SampleSlope(3); /* worst case sample jitter */
504
505 return(SUCCESS);
506 }
507
508 /*
509  * lm_tmg_set_frequency()
510  *
511  * This function sets the clock and starts timer using
512  * Counter 1. This timer is used along with lock detect
513  * to decide if clock frequency has stabilized. Counter 1
514  * is loaded with 1750 - 1 and counts at 3.75 MHz, resulting
515  * in a time delay of 1 ns.
516  *
517  * Inputs: none
518  * Outputs: returns SUCCESS or FAILURE
519  */
520 static int tmg_was_changed;
521 lm_tmg_set_frequency(a, k, source)
522     u_char a;
523     u_char k;
524     u_char source;
525 {
526     static u_char tmg_last_a = 0;
527     int reala, realk;
528     int maxtries;
529
530     if((source == EXT0) || (source == EXT1))
531     /* can't set frequency with clock on */
532     if(lm_tmg_clockoff() != SUCCESS)
533         return(FAILURE);
534     tmgptr->clock_select = source;
535     tmg_was_changed = FALSE;
536     if(islow())
537     {
538         lm_queue_message(ERROR_MSG, "External clock is slow");
539         return FAILURE;
540     }
541     else
542         return SUCCESS;
543 }
544
545 reala = 257 - a;
546 realk = 257 - k;
547
548 if((reala < 128) || (realk == 0) || (source > 2)) {
549     lm_queue_message(ERROR_MSG, "illegal value for a,k,source: a = %d, k = %d, source = %d",
550         a, k, source);
551     return(FAILURE);
552 }
553
554 /* can't set frequency with clock on */
555 if(lm_tmg_clockoff() != SUCCESS)
556     return(FAILURE);
557
558 if(a == tmg_last_a) /* pll does not need any lock time */
559 {
560     tmg_was_changed = FALSE;
561     tmgptr->pll_divisor = k;
562     tmgptr->clock_select = source;
563     return(SUCCESS);
564 }
565
566 tmgptr->clock_select = 1; /* don't know what freq, but its square */
567 if(lm_tmg_clockon() != SUCCESS)
568     return(FAILURE);
569
570 tmgptr->ctc_intr_clear = 0;
571 tmgptr->ctc_register[3].value = 0x10; /* setup mode 0 */
572 tmgptr->ctc_register[0].value = zero; /* lab for max count */
573 tmgptr->ctc_register[0].value = zero; /* max for max count */
574
575 /* since GATE0 has no effect on OUT0, OUT0 should be low */
576 /* and furthermore, Counter0 should not be counting */
577
578 /* at this point, merely verify that Counter0 output is low */
579 maxtries = 10000;
580 while(1)
581 {
582     tmgptr->ctc_register[3].value = LATCH_CNTR_0;
583     if(!((tmgptr->ctc_register[0].value & 0x80)))
584         break;
585     if(--maxtries == 0)
586     {
587         lm_queue_message(ERROR_MSG, "CTC failure\n");
588         return FAILURE;
589     }
590 }
591
592 /* turn the clock off to keep the PACs & PLLs happy */
593 if(lm_tmg_clockoff() != SUCCESS) /* shut off clock while output true */
594     return(FAILURE);
595
596 tmgptr->ctc_register[3].value = 0x070; /* setup mode 0 */
597 tmgptr->ctc_register[1].value = 0xa5; /* lab of 1750 - 1 */
598 tmgptr->ctc_register[1].value = 0x0e; /* max of 1750 - 1 */
599
600

```

| | | | | |
|--|--|-----------------------------------|--------------------|-----------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_run.c | DATE 5/23/89 | PAGE # 6/203 |
| | | | TIME 4:41:35 pm | |
| LINE # | SOURCE TEXT | | | |
| 601 | /* timer is now running, using Counter 1 */ | | | |
| 602 | tmg_last_s = s; | | | |
| 603 | tmg_s_was_changed = TRUE; | | | |
| 604 | tmgptr->pll_rate = s; | | | |
| 605 | tr->pll_divisor = k; | | | |
| 606 | tmg->clock_select = source; | | | |
| 607 | return(SUCCESS); | | | |
| 608 | } | | | |
| 609 | | | | |
| 610 | /* | | | |
| 611 | lm_tmg_check_locked() | | | |
| 612 | /* | | | |
| 613 | This function returns SUCCESS if PLL lock indicator is true | | | |
| 614 | and 1 ms has expired since changing its frequency, as | | | |
| 615 | indicated by Counter 1. Counter 1 is setup by | | | |
| 616 | lm_tmg_set_frequency(). | | | |
| 617 | */ | | | |
| 618 | lm_tmg_check_locked() | | | |
| 619 | { | | | |
| 620 | if(!tmg_s_was_changed && tmgptr->pll_locked) /* no need to wait */ | | | |
| 621 | return(SUCCESS); | | | |
| 622 | /* when Counter 1 output goes true, desired time delay has passed */ | | | |
| 623 | tmgptr->ctc_register[3].value = LATCH_CNTR_1_STATUS; | | | |
| 624 | if(((tmgptr->ctc_register[1].value & 0x0c0) == 0x080) && tmgptr->pll_locked) | | | |
| 625 | return(SUCCESS); /* null flag not set and output true */ | | | |
| 626 | else | | | |
| 627 | return(FAILURE); | | | |
| 628 | } | | | |
| 629 | | | | |
| 630 | | | | |
| 631 | | | | |
| 632 | | | | |
| 633 | /* | | | |
| 634 | lm_tmg_measure_clock() | | | |
| 635 | /* | | | |
| 636 | This function measures the frequency of either external clock. | | | |
| 637 | */ | | | |
| 638 | Inputs: clock type (external 0 or 1), pointer to clock period in ps | | | |
| 639 | Outputs: clock period is assigned, function returns SUCCESS | | | |
| 640 | or FAILURE. | | | |
| 641 | */ | | | |
| 642 | lm_tmg_measure_clock(clocktype, periodptr) | | | |
| 643 | u_char clocktype; | | | |
| 644 | u_long *periodptr; | | | |
| 645 | { | | | |
| 646 | int i, saveintenable; | | | |
| 647 | u_long least, most, cnt1[3], cnt2[3], start; | | | |
| 648 | if(lm_tmg_clockoff() != SUCCESS) | | | |
| 649 | return(FAILURE); | | | |
| 650 | if((clocktype != EXT0) && (clocktype != EXT1)) { | | | |
| 651 | lm_queue_message(ERROR_MSG, "invalid clock type (%d)", | | | |
| 652 | clocktype); | | | |
| 653 | return(FAILURE); | | | |
| 654 | tmgptr->clock_select = clocktype; | | | |
| 655 | if(lm_tmg_clockon() != SUCCESS) | | | |
| 656 | return(FAILURE); | | | |
| 657 | if(isalow()) { | | | |
| 658 | lm_queue_message(ERROR_MSG, "clock is slow"); | | | |
| 659 | return(FAILURE); | | | |
| 660 | } | | | |
| 661 | saveintenable = tmgptr->ctc_intr_enable; | | | |
| 662 | tmgptr->ctc_intr_enable = 0; | | | |
| 663 | tmgptr->ctc_intr_clear1 = 0; /* output false */ | | | |
| 664 | tmgptr->ctc_register[3].value = CNTR0CM; /* setup 0 */ | | | |
| 665 | tmgptr->ctc_register[0].value = 0x00; /* lab of 1000 */ | | | |
| 666 | tmgptr->ctc_register[0].value = 0x01; /* mab of 1000 */ | | | |
| 667 | tmgptr->ctc_register[3].value = CNTR1CM; /* setup 1,2 */ | | | |
| 668 | tmgptr->ctc_register[1].value = zero; | | | |
| 669 | tmgptr->ctc_register[2].value = zero; | | | |
| 670 | tmgptr->ctc_register[3].value = CNTR2CM; | | | |
| 671 | tmgptr->ctc_register[2].value = zero; | | | |
| 672 | tmgptr->ctc_register[2].value = zero; | | | |
| 673 | tmgptr->ctc_intr_clear1 = 1; /* let her rip */ | | | |
| 674 | start = lm_time(); | | | |
| 675 | while(!tmgptr->ctc_intr) | | | |
| 676 | if((lm_time() - start) > 1000) | | | |
| 677 | { | | | |
| 678 | tmgptr->ctc_intr_clear1 = 0; | | | |
| 679 | tmgptr->ctc_intr_enable = saveintenable; | | | |
| 680 | lm_queue_message(ERROR_MSG, "timeout, no CTC interrupt"); | | | |
| 681 | return(FAILURE); | | | |
| 682 | } | | | |
| 683 | tmgptr->ctc_register[3].value = CNTR12LATCH; /* latch count/status */ | | | |
| 684 | for(i = 0; i < 3; i++) | | | |
| 685 | cnt1[i] = tmgptr->ctc_register[1].value & 0x0ff; | | | |
| 686 | for(i = 0; i < 3; i++) | | | |
| 687 | cnt2[i] = tmgptr->ctc_register[2].value & 0x0ff; | | | |
| 688 | if(cnt1[0] & 0x40) /* check null flag */ | | | |
| 689 | least = 0; | | | |
| 690 | else | | | |
| 691 | { | | | |
| 692 | least = cnt1[1] + (cnt1[2] << 8); | | | |
| 693 | if(least == 0) | | | |
| 694 | least = 1; | | | |
| 695 | else | | | |
| 696 | least = 0x100011 - least; | | | |
| 697 | } | | | |
| 698 | if(cnt2[0] & 0x40) /* check null flag */ | | | |
| 699 | most = 0; | | | |
| 700 | else | | | |
| 701 | { | | | |
| 702 | most = cnt2[1] + (cnt2[2] << 8); | | | |
| 703 | if(most == 0) | | | |
| 704 | most = 1; | | | |
| 705 | else | | | |
| 706 | most = 0x100011 - most; | | | |
| 707 | } | | | |
| 708 | *periodptr = (((most << 16) - least) * 1000) / 40; | | | |
| 709 | | | | |
| 710 | | | | |
| 711 | | | | |
| 712 | | | | |
| 713 | | | | |
| 714 | | | | |
| 715 | | | | |
| 716 | | | | |
| 717 | | | | |
| 718 | | | | |
| 719 | | | | |
| 720 | | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_run.c | DATE 5/23/89 TIME 4:41:35 pm | PAGE # 7/204 |
|--|--|-----------------------------------|---------------------------------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 721 | tmgptr->ctr_intr_clearl = 0; | | | |
| 722 | tmgptr->ctr_intr_enable = saveintrenable; | | | |
| 723 | return(SUCCESS); | | | |
| 724 | } | | | |
| 725 | } | | | |
| 726 | salow() | | | |
| 727 | { | | | |
| 728 | register long counter; | | | |
| 729 | u_long start; | | | |
| 730 | if(lm_tmg_clockon() != SUCCESS) | | | |
| 731 | return(1); /* same as slow clock */ | | | |
| 732 | start = lm_time(); | | | |
| 733 | do | | | |
| 734 | { | | | |
| 735 | if((lm_time() - start) > TIMEOUT) | | | |
| 736 | { | | | |
| 737 | lm_queue_message(ERROR_MSG, "salow(): error in initialization\n"); | | | |
| 738 | return FAILURE; | | | |
| 739 | } | | | |
| 740 | tmgptr->slow_clock_clearl = 0; /* clear it */ | | | |
| 741 | tmgptr->slow_clock_clearl = 1; /* unclear it */ | | | |
| 742 | while(tmgptr->slow_clock); /* it set, repeat */ | | | |
| 743 | } | | | |
| 744 | /* wait about 25 msec to see if slow clock */ | | | |
| 745 | counter = 49; /* delay = 500 ns * (49 + 1) = 25000 ns */ | | | |
| 746 | while(--counter) | | | |
| 747 | { | | | |
| 748 | return(tmgptr->slow_clock); | | | |
| 749 | } | | | |
| 750 | } | | | |
| 751 | /* | | | |
| 752 | int lm_tmg_clockoff(void) | | | |
| 753 | /* | | | |
| 754 | shuts clock off | | | |
| 755 | Input: nothing | | | |
| 756 | Output: SUCCESS or FAILURE | | | |
| 757 | */ | | | |
| 758 | int lm_tmg_clockoff() | | | |
| 759 | { | | | |
| 760 | u_long start; | | | |
| 761 | tmgptr->clock_enable = 0; | | | |
| 762 | start = lm_time(); | | | |
| 763 | while(tmgptr->clock_on) | | | |
| 764 | { | | | |
| 765 | if((lm_time() - start) > 10) | | | |
| 766 | { | | | |
| 767 | if((tmgptr->clock_select > 2) | | | |
| 768 | tmgptr->clock_sync_clearl = 0; /* must be external clock */ | | | |
| 769 | break; | | | |
| 770 | } | | | |
| 771 | } | | | |
| 772 | if(tmgptr->clock_on) | | | |
| 773 | { | | | |
| 774 | lm_queue_message(ERROR_MSG, "can't turn off clock"); | | | |
| 775 | return(FAILURE); | | | |
| 776 | } | | | |
| 777 | tmgptr->clock_sync_clearl = 0; | | | |
| 778 | return(SUCCESS); | | | |
| 779 | } | | | |
| 780 | /* | | | |
| 781 | int lm_tmg_clockon(void) | | | |
| 782 | /* | | | |
| 783 | turns clock on | | | |
| 784 | Input: nothing | | | |
| 785 | Output: SUCCESS or FAILURE | | | |
| 786 | */ | | | |
| 787 | int lm_tmg_clockon() | | | |
| 788 | { | | | |
| 789 | u_long start; | | | |
| 790 | if(lm_tmg_clockoff() != SUCCESS) | | | |
| 791 | return FAILURE; | | | |
| 792 | tmgptr->clock_sync_clearl = 1; | | | |
| 793 | tmgptr->clock_enable = 1; | | | |
| 794 | start = lm_time(); | | | |
| 795 | while(!tmgptr->clock_on) | | | |
| 796 | { | | | |
| 797 | if((lm_time() - start) > 10) | | | |
| 798 | { | | | |
| 799 | lm_queue_message(ERROR_MSG, "can't turn on clock"); | | | |
| 800 | return(FAILURE); | | | |
| 801 | } | | | |
| 802 | } | | | |
| 803 | return(SUCCESS); | | | |
| 804 | } | | | |
| 805 | /* | | | |
| 806 | lm_tmg_set_falling_sample(count) | | | |
| 807 | /* | | | |
| 808 | This function sets the sample width counter with the | | | |
| 809 | specified count after checking it against bounds. | | | |
| 810 | Inputs: count (number of clock cycles - 1 for width) | | | |
| 811 | Outputs: function returns SUCCESS or FAILURE | | | |
| 812 | */ | | | |
| 813 | lm_tmg_set_falling_sample(count) | | | |
| 814 | { | | | |
| 815 | u_long count; | | | |
| 816 | if(count > 255) | | | |
| 817 | { | | | |
| 818 | lm_queue_message(ERROR_MSG, "value too large (td)", | | | |
| 819 | count); | | | |
| 820 | return(FAILURE); | | | |
| 821 | } | | | |
| 822 | else | | | |
| 823 | { | | | |
| 824 | tmgptr->sample_width = count; | | | |
| 825 | return(SUCCESS); | | | |
| 826 | } | | | |
| 827 | } | | | |
| 828 | /* | | | |
| 829 | lm_tmg_set_rising_sample(range, threshold) | | | |
| 830 | /* | | | |
| 831 | This function sets the comparator threshold used to | | | |
| 832 | determine rising edge of SAMPLE. Threshold is checked | | | |
| 833 | against bounds. | | | |
| 834 | Inputs: sample ramp and threshold value | | | |
| 835 | Outputs: function returns SUCCESS or FAILURE | | | |
| 836 | */ | | | |
| 837 | lm_tmg_set_rising_sample(range, threshold) | | | |
| 838 | { | | | |
| 839 | /* | | | |
| 840 | This function sets the comparator threshold used to | | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_run.c

DATE

5/23/89

PAGE #

TIME

4:41:35 pm

8/205

```

LINE #          SOURCE TEXT
841  /*
842  lm_tmg_set_rising_sample(range, threshold)
843  u_long range,
844  u_long threshold;
845  {
846      if((range > 3) || (threshold > 255) ||
847          (threshold < calib.SampleMinThresh(range))) {
848          lm_queue_message(ERROR_MSG, "range too large (%d) OR threshold out of range (%d)",
849                          range, threshold);
850          return(FAILURE);
851      }
852      else
853      {
854          tmgptr->sample_delay = threshold;
855          tmgptr->sample_delay_range = range;
856          return(SUCCESS);
857      }
858  }
859
860  /*
861  lm_tmg_set_edge_settings(period, thresholds)
862  *
863  *   This function sets the comparator thresholds used
864  *   for the six timing edges and the sample ramp trigger.
865  *   Thresholds are checked against bounds.
866  *
867  *   Inputs: logical clock period in ps, and threshold array
868  *   Outputs: function returns SUCCESS or FAILURE
869  */
870  lm_tmg_set_edge_settings(period, thresholds)
871  u_long period,
872  u_long thresholds[];
873  {
874      int ramp;
875
876      for(ramp = 0; ramp < 4; ramp++)
877          if(calib.EdgeMaxDelay[ramp] > period)
878              return lm_tmg_set_ramp_edge_settings(ramp, thresholds);
879
880      lm_queue_message(ERROR_MSG, "period too large, period = %d, EdgeMaxDelay[3] = %d",
881                      period, calib.EdgeMaxDelay[3]);
882      return(FAILURE);
883  }
884
885  lm_tmg_set_ramp_edge_settings(ramp, thresholds)
886  register u_long ramp;
887  u_long thresholds[];
888  {
889      register u_long edge;
890
891      for(edge = 0; edge < 6; edge++)
892          if((thresholds[edge] > 255) ||
893              (thresholds[edge] < calib.EdgeMinThresh(ramp)[edge])) {
894              lm_queue_message(ERROR_MSG, "threshold for edge %d is out of range (%d)",
895                              edge, thresholds[edge]);
896              return(FAILURE);
897          }
898      else
899          tmgptr->edge_delay[edge].delay = thresholds[edge];
900
901      #ifdef possible_bug
902      if((thresholds[6] > 255) || (thresholds[6] < calib.EdgeMinThresh(ramp)[6]))
903          #else possible_bug
904      if((thresholds[6] > 255) || (thresholds[6] < calib.Edge7MinThresh(ramp)))
905      #endif possible_bug
906          return(FAILURE);
907      else
908          tmgptr->sample_trigger_threshold = thresholds[6]; /* edge "7" */
909
910      tmgptr->edge_delay_range = ramp;
911      return(SUCCESS);
912  }
913
914  /*
915  int lm_tmg_play(timeout)
916  *
917  *   Initiates pattern presentation
918  *
919  *   Inputs: none
920  *   Returns: SUCCESS or FAILURE
921  */
922  int lm_tmg_play(timeout)
923  u_long timeout;
924  {
925      if(lm_tmg_initiate_play() != SUCCESS)
926          return(FAILURE);
927      return(lm_tmg_complete_play(timeout));
928  }
929
930  /*
931  int lm_tmg_initiate_play(void)
932  *
933  *   This routine turns clocks on and starts a presentation.
934  *
935  *   Inputs: none
936  *   Outputs: function returns SUCCESS or FAILURE
937  */
938  int lm_tmg_initiate_play()
939  {
940      if(tmgptr->lase_intr) /* is error on bus? */
941          if(tmgptr->lase_intr) /* really? */
942          {
943              lm_queue_message(ERROR_MSG, "lm_tmg_initiate_play: error on bus");
944              return(FAILURE);
945          }
946      if(tmgptr->backplane_mode) /* in PLAY mode? */
947          lm_queue_message(ERROR_MSG, "lm_tmg_initiate_play: already in PLAY mode");
948      return(FAILURE);
949  }
950  if(lm_tmg_clockoff() != SUCCESS)
951      return(FAILURE);
952  if(lm_tmg_clockon() != SUCCESS)
953      return(FAILURE);
954
955  play_completed_flag = 0;
956  post_end_of_play();
957  tmgptr->pattern_intr_enable = 0; /* just in case */
958  tmgptr->pattern_intr_enable = 1; /* enable EOP interrupt */
959  tmgptr->start_pattern_play = 1; /* start presentation */
960  return(SUCCESS);

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_run.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:35 pm | 9/206 |

```

LINE #          SOURCE TEXT
961  }
962
963
964  /*      int lm_tag_complete_play(u_long timeout)
965  *
966  *      This function waits for a presentation to complete by
967  *      waiting for the EOP interrupt from the Timing Generator.
968  *
969  *      Inputs: timeout is ms
970  *      Outputs: function returns SUCCESS or FAILURE
971  */
972  int lm_tag_complete_play(timeout)
973  u_long timeout;
974  {
975      u_long start;
976
977      #ifdef DIAGS
978      #ifdef MODELER
979      int err;
980
981      while (! play_completed_flag)
982      {
983          ac_spend( play_semaphore, lm_number_of_ticks( timeout ), &err );
984          if( err == VRTX_TIMEOUT )
985          {
986              lm_tag_abort_play();
987              tmgptr->pattern_intr_enable = 0;      /* clear EOP interrupt */
988              post_end_of_play=0;
989              return( FAILURE );
990          }
991          if( err != VRTX_OK )
992          {
993              post_end_of_play=0;
994              printf("Error in lm_tag_complete_play(), error code %x\n",err);
995              tmgptr->pattern_intr_enable = 0;      /* clear EOP interrupt */
996              return(FAILURE);
997          }
998      }
999      post_end_of_play=0;
1000      tmgptr->pattern_intr_enable = 0;      /* clear EOP interrupt */
1001      return(SUCCESS);
1002      #endif
1003      #else DIAGS
1004      start = lm_time();
1005      while(!TMC_INT)
1006      {
1007          if((lm_time() - start) > timeout)      /* wait for interrupt */
1008          {
1009              lm_tag_abort_play();
1010              tmgptr->pattern_intr_enable = 0;      /* clear EOP interrupt */
1011              return(FAILURE);
1012          }
1013          tmgptr->pattern_intr_enable = 0;      /* clear EOP interrupt */
1014          return(SUCCESS);
1015      }
1016      #endif DIAGS
1017  }
1018  /*      int lm_tag_abort_play()
1019  *
1020  *      INPUT: none
1021  *      OUTPUT: returns SUCCESS or FAILURE
1022  *      DESCRIPTION: Aborts pattern play by asserting and then removing
1023  *      the error line.
1024  */
1025  int
1026  lm_tag_abort_play()
1027  {
1028      u_long start;
1029      int stuck, returncode;
1030      int save_lane_intr_enable;
1031
1032      save_lane_intr_enable = tmgptr->lane_intr_enable;
1033
1034      stuck = FALSE;
1035      returncode = SUCCESS;
1036
1037      if(!tmgptr->backplane_mode)      /* if ACCESS mode */
1038      {
1039          tmgptr->pattern_intr_enable = 0;      /* clear EOP interrupt */
1040          if(lm_tag_clockoff() != SUCCESS)
1041          {
1042              lm_queue_message(ERROR_MSG, "lm_tag_complete_play: clockoff returned error");
1043              return(FAILURE);
1044          }
1045      }
1046      else
1047      {
1048          disable_mod_err();
1049          if(tmgptr->lane_enable)
1050          {
1051              if(tmgptr->lane_enable & ~tmgptr->lane_intr)
1052              {
1053                  start = lm_time();
1054                  tmgptr->lane_intr_enable = 0;
1055                  tmgptr->backplane_error = 1;      /* Assert error line */
1056                  while(tmgptr->clock_on)      /* Wait for clock to stop */
1057                  {
1058                      if((lm_time() - start) > 10)      /* 10 ms timeout */
1059                      {
1060                          stuck = TRUE;
1061                          lm_queue_me_ "(ERROR_MSG, "lm_tag_abort_play: PAC did not respond to ERROR");
1062                          break;
1063                      }
1064                      tmgptr->backplane_error = 0;      /* Deassert error line */
1065                  }
1066              }
1067          }
1068          if(stuck)
1069          {
1070              tmgptr->abort_pattern_play = 1;      /* last resort */
1071              start = lm_time();
1072              while(tmgptr->clock_on)
1073              {
1074                  if((lm_time() - start) > 10)
1075                  {
1076                      returncode = FAILURE;
1077                      lm_queue_message(ERROR_MSG, "lm_tag_abort_play: TMG state machine did not quit");
1078                      lm_queue_message(ERROR_MSG, "mode = %d",
1079                                      tmgptr->backplane_mode);
1080                      lm_queue_message(ERROR_MSG, "clock_on = %d",
1081                                      tmgptr->clock_on);
1082                      lm_queue_message(ERROR_MSG, "errors = %x",
1083                                      tmgptr->lane_intr);
1084                  }
1085              }
1086          }
1087      }
1088  }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_run.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:41:35 pm | 10/207 |

| LINE # | SOURCE TEXT |
|--------|--|
| 1081 | } |
| 1082 | } |
| 1083 | } |
| 1084 | /* Finish error signal out of pipeline */ |
| 1085 | if(lm_tmg_clockoff() != SUCCESS) /* Turn off clock */ |
| 1086 | returncode = FAILURE; |
| 1087 | if(lm_tmg_clockon() != SUCCESS) /* Turn on clock */ |
| 1088 | returncode = FAILURE; |
| 1089 | if(lm_tmg_clockoff() != SUCCESS) /* Turn off clock */ |
| 1090 | returncode = FAILURE; |
| 1091 | |
| 1092 | tmgptr->lase_intr_enable = save_lase_intr_enable; |
| 1093 | |
| 1094 | enable_mod_err(); |
| 1095 | return(returncode); |
| 1096 | } |
| 1097 | |
| 1098 | /* |
| 1099 | lm_tmg_set_sample_trigger_mode(mode) |
| 1100 | */ |
| 1101 | /* |
| 1102 | Selects either normal or early sample trigger mode. |
| 1103 | */ |
| 1104 | /* |
| 1105 | Input: mode |
| 1106 | Output: returns SUCCESS or FAILURE if mode is not 0 or 1 |
| 1107 | */ |
| 1108 | lm_tmg_set_sample_trigger_mode(mode) |
| 1109 | u_long mode; |
| 1110 | { |
| 1111 | if(mode > 1) |
| 1112 | { |
| 1113 | lm_queue_message(ERROR_MSG, "lm_tmg_set_sample_trigger_mode: invalid mode"); |
| 1114 | return FAILURE; |
| 1115 | else |
| 1116 | { |
| 1117 | tmgptr->sample_mode = mode; |
| 1118 | return SUCCESS; |
| 1119 | } |
| 1120 | } |
| 1121 | |
| 1122 | |

| Copyright 1989 Logic Modeling Systems | | HEADER FILE diags/tmg_run.h | DATE 5/23/89 TIME 4:41:37 pm | PAGE # 1/208 |
|--|---|--------------------------------|---------------------------------------|-----------------|
| LINE # | HEADER TEXT | | | |
| 1 | /* SCCS ID: tmg_run.h rev 3.1, 4/24/89 at 07:51:00 */ | | | |
| 2 | | | | |
| 3 | #define N_MIN 128 | | | |
| 4 | #define N_MAX 256 | | | |
| 5 | #define T_REF 8533333.3 /* reference period in ps */ | | | |
| 6 | #define MAX_PERIOD 6667746 /* 1 / 150 KHz in ps */ | | | |
| 7 | | | | |
| 8 | #define MIN_PERIOD 40000.0 /* actually our approx of 150 MHz */ | | | |
| 9 | #define DEAD_TIME 15000 /* 1 / 25 MHz in ps */ | | | |
| 10 | #define LATCH_CNTR_0 /* guess in ps */ | | | |
| 11 | #define LATCH_CNTR_0 0x0e2 /* latch count in Counter 0 */ | | | |
| 12 | #endif | | | |
| 13 | #define LATCH_CNTR_1 | | | |
| 14 | #define LATCH_CNTR_1 0x0c4 /* latch count in Counter 1 */ | | | |
| 15 | #endif | | | |
| 16 | #define LATCH_CNTR_1_STATUS 0xe4 /* latch status only */ | | | |
| 17 | | | | |
| 18 | #define EXT0 3 /* register value to select ext clock 0 */ | | | |
| 19 | #define EXT1 4 /* register value to select ext clock 1 */ | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM diags/tmg_util.c | DATE 5/23/89 | PAGE # 1/209 |
|--|---|------------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCOS ID: tmg_util.c rev. 3.1, 4/26/89 at 07:51:03 */ | | | |
| 2 | /* | | | |
| 3 | * tmg_util.c | | | |
| 4 | * A collection of small utility routines for the Timing | | | |
| 5 | * Generator Diagnostics and Calibration | | | |
| 6 | * | | | |
| 7 | * | | | |
| 8 | * | | | |
| 9 | * | | | |
| 10 | * | | | |
| 11 | * | | | |
| 12 | #include "common.h" | | | |
| 13 | #include "moduler_extn.h" | | | |
| 14 | #include "mod_def.h" | | | |
| 15 | #include "tmg.h" | | | |
| 16 | #include "tmg_def.h" | | | |
| 17 | #include "vtr.h" | | | |
| 18 | #include "tmg_extn.h" | | | |
| 19 | #ifdef DIAGS | | | |
| 20 | #include "diag_setjmp.h" | | | |
| 21 | #endif DIAGS | | | |
| 22 | #include "istr.h" | | | |
| 23 | static int zero = 0; | | | |
| 24 | /* | | | |
| 25 | * u_long tmg_measure_freq(void) | | | |
| 26 | * This routine measures clock frequency of the selected clock. | | | |
| 27 | * It counts how many unknown clocks occur within 10,000 clocks | | | |
| 28 | * of 3.75 MHz. If the unknown frequency is as low as 150 KHz, | | | |
| 29 | * there can be as many as 250,000 reference clocks, thus the | | | |
| 30 | * need for a 32 bit count of reference clocks. | | | |
| 31 | * Inputs: none | | | |
| 32 | * Returns: number of clocks (3.75 MHz, 256.7 ns period) | | | |
| 33 | */ | | | |
| 34 | u_long tmg_measure_freq(soclockptr) | | | |
| 35 | u_long *soclockptr, | | | |
| 36 | { | | | |
| 37 | int i; | | | |
| 38 | u_long least, most, cnt1[3], cnt2[3], start; | | | |
| 39 | if(tmg_clockoff() != SUCCESS) | | | |
| 40 | { | | | |
| 41 | lm_error("Measure Freq: could not turn clock off.\n"); | | | |
| 42 | return(FAILURE); | | | |
| 43 | } | | | |
| 44 | if(tmg_clockon() != SUCCESS) | | | |
| 45 | { | | | |
| 46 | lm_error("Measure Freq: could not turn clock on.\n"); | | | |
| 47 | lm_message("Please check the external clock connection.\n"); | | | |
| 48 | return(FAILURE); | | | |
| 49 | } | | | |
| 50 | if(tmg_islow()) | | | |
| 51 | { | | | |
| 52 | lm_warning("The frequency is too slow.\n"); | | | |
| 53 | lm_warning("==> Please check the external clock connection.\n"); | | | |
| 54 | lm_warning("==> Allowed range is 150 KHz to 25 MHz, 0-4 V p-p.\n"); | | | |
| 55 | } | | | |
| 56 | tmgptr->ctc_intr_clearl = 0; /* output false */ | | | |
| 57 | tmgptr->ctc_register[3].value = CNTR0CN, /* setup 0 */ | | | |
| 58 | tmgptr->ctc_register[0].value = 0x10, /* 1st of 10000 */ | | | |
| 59 | tmgptr->ctc_register[0].value = 0x17, /* 2nd of 10000 */ | | | |
| 60 | tmgptr->ctc_register[3].value = CNTR1CN, /* setup 1,2 */ | | | |
| 61 | tmgptr->ctc_register[1].value = zero; | | | |
| 62 | tmgptr->ctc_register[1].value = zero; | | | |
| 63 | tmgptr->ctc_register[3].value = CNTR2CN, | | | |
| 64 | tmgptr->ctc_register[2].value = zero; | | | |
| 65 | tmgptr->ctc_register[2].value = zero; | | | |
| 66 | tmgptr->ctc_intr_clearl = 1; /* let her rip */ | | | |
| 67 | start = lm_time(); | | | |
| 68 | while(!tmgptr->ctc_intr) /* wait for finish */ | | | |
| 69 | { | | | |
| 70 | if((lm_time() - start) > 5000) | | | |
| 71 | { | | | |
| 72 | lm_error("Measure Freq: timeout error waiting for CTC.\n"); | | | |
| 73 | goto failure; | | | |
| 74 | } | | | |
| 75 | tmgptr->ctc_register[3].value = CNTR2LATCH, /* latch count/status */ | | | |
| 76 | for(i = 0; i < 3; i++) | | | |
| 77 | cnt1[i] = tmgptr->ctc_register[i].value & 0x0ff; | | | |
| 78 | for(i = 0; i < 3; i++) | | | |
| 79 | cnt2[i] = tmgptr->ctc_register[i].value & 0x0ff; | | | |
| 80 | if(cnt1[0] & 0x40) /* check null flag */ | | | |
| 81 | { | | | |
| 82 | least = 0; | | | |
| 83 | { | | | |
| 84 | least = cnt1[1] + (cnt1[2] << 8); | | | |
| 85 | if(least == 0) | | | |
| 86 | least = 1; | | | |
| 87 | else | | | |
| 88 | least = 0x100011 - least; | | | |
| 89 | } | | | |
| 90 | if(cnt2[0] & 0x40) /* check null flag */ | | | |
| 91 | { | | | |
| 92 | most = 0; | | | |
| 93 | { | | | |
| 94 | most = cnt2[1] + (cnt2[2] << 8); | | | |
| 95 | if(most == 0) | | | |
| 96 | most = 1; | | | |
| 97 | else | | | |
| 98 | most = 0x100011 - most; | | | |
| 99 | } | | | |
| 100 | *soclockptr = (most << 16) + least; | | | |
| 101 | if(tmg_clockoff() != SUCCESS) | | | |
| 102 | { | | | |
| 103 | lm_error("Measure Freq: could not turn clock off before successful exit.\n"); | | | |
| 104 | return(FAILURE); | | | |
| 105 | } | | | |
| 106 | return(SUCCESS); | | | |
| 107 | failure: | | | |
| 108 | if(tmg_clockoff() != SUCCESS) | | | |
| 109 | { | | | |
| 110 | lm_error("Measure Freq: could not turn clock off before successful exit.\n"); | | | |
| 111 | return(FAILURE); | | | |
| 112 | } | | | |
| 113 | return(SUCCESS); | | | |
| 114 | failure: | | | |
| 115 | if(tmg_clockoff() != SUCCESS) | | | |
| 116 | { | | | |
| 117 | lm_error("Measure Freq: could not turn clock off before successful exit.\n"); | | | |
| 118 | return(FAILURE); | | | |
| 119 | } | | | |
| 120 | return(SUCCESS); | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_util.c

DATE 5/23/89
TIME 4:41:37 pm

PAGE #
2/210

```

121  ln_error("Measure Freq: could not turn clock off before aborted exit.\n");
122  return(FAILURE);
123  }
124  return FAILURE;
125  }
126
127  tmg_measure_period()
128  {
129      u_long num_of_clocks;
130
131      /* Measure the clock speed */
132      if (tmg_measure_freq(&num_of_clocks) != SUCCESS)
133      {
134          (void)ln_error("Clock frequency measurement failed.\n");
135          return 0;
136      }
137      /* return pattern clock period in picoseconds */
138      return ((num_of_clocks * 101) / 61);
139  }
140
141
142  /*
143   * int tmg_isalow(void)
144   * Checks if clock is slow or not
145   * Inputs: none
146   * Outputs: 1 if slow, 0 if not
147   */
148  int tmg_isalow()
149  {
150      tmgptr->slow_clock_clear1 = 0;
151      tmgptr->slow_clock_clear1 = 1;
152      ln_delay(10);
153      return(tmgptr->slow_clock);
154  }
155
156  /*
157   * int tmg_toggle(int edge, int recount, int mode)
158   * This routine checks to see if the specified edge's
159   * calibration flip flop can toggle with the current
160   * setup (delay line settings, threshold, clock rate, etc).
161   * In particular, the proper test mode, if any, must be setup.
162   * Inputs: the edge number to toggle 0..3
163   *         a repeat count, and a mode bit
164   * Returns: number of toggles, or -1 on error
165   */
166
167  int tmg_toggle(edge, recount, mode)
168  int edge, recount, mode;
169  {
170      int current, mask, previous, timestoggled;
171
172      if((edge < 0) || (edge > 5))
173      {
174          sys_out("tmg_toggle: invalid edge specified\n");
175          return(-1);
176      }
177
178      if(tmg_calclear() != SUCCESS)
179      {
180          sys_out("tmg_toggle: cal flip flops did not clear\n");
181          return(-1);
182      }
183
184      timestoggled = 0;
185      previous = 0;
186      mask = 1 << edge;
187
188      if(mode == TOGGLEMODE)
189      {
190          while(recount-- > 0)
191          {
192              if(tmg_play(TIMEOUT) != SUCCESS)
193              {
194                  sys_out("tmg_toggle: tmg_play(TIMEOUT) returned error\n");
195                  return(-1);
196              }
197              if((current = tmgptr->edge_cal & mask) != previous)
198              {
199                  previous = current;
200                  timestoggled++;
201              }
202              else
203                  break;
204          }
205      }
206      else
207      {
208          if(mode == NOTOGGLEMODE)
209          {
210              while(recount-- > 0)
211              {
212                  if(tmg_play(TIMEOUT) != SUCCESS)
213                  {
214                      sys_out("tmg_toggle: tmg_play(TIMEOUT) returned error\n");
215                      return(-1);
216                  }
217                  if((current = tmgptr->edge_cal & mask) != previous)
218                  {
219                      timestoggled++;
220                      break;
221                  }
222              }
223          }
224          else
225              return(-1);
226      }
227
228      return(timestoggled);
229  }
230
231  /*
232   * int tmg_toggle_all(count, no_toggles)
233   * This function checks all edge calibration flip flops for toggling.
234   * This should be used when searching through clock periods to find
235   * points along ramps.
236   * Inputs: count, array of toggles for each edge
237   * Outputs: array of toggles is assigned, function returns SUCCESS
238   *          or FAILURE
239   */
240  int tmg_toggle_all(count, no_toggles)
241  int count, no_toggles[];

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_util.c

DATE

5/23/89

PAGE #

TIME

4:41:37 pm

3/211

| LINE # | SOURCE TEXT |
|--------|---|
| 241 | { |
| 242 | u_char data; |
| 243 | int i; |
| 244 | if(tmg_effclear() != SUCCESS) |
| 245 | { |
| 246 | lm_error("tmg_toggle_all: cal flip flops dis not clear\n"); |
| 247 | return(FAILURE); |
| 248 | } |
| 249 | for(i = 0; i < 6; i++) |
| 250 | { |
| 251 | no_toggles[i] = 0; |
| 252 | data = 0; |
| 253 | } |
| 254 | while(count--) |
| 255 | { |
| 256 | if(tmg_play(TIMEOUT) != SUCCESS) |
| 257 | { |
| 258 | lm_error("tmg_toggle_all: tmg_play(TIMEOUT) returned error\n"); |
| 259 | return(FAILURE); |
| 260 | } |
| 261 | data = tmgptr->edge_cal ^ data; |
| 262 | for(i = 0; i < 6; i++) |
| 263 | { |
| 264 | no_toggles[i] += (data >> i) & 1; |
| 265 | } |
| 266 | } |
| 267 | return(SUCCESS); |
| 268 | } |
| 269 | |
| 270 | /* int tmg_init(void) |
| 271 | /* |
| 272 | This function initializes the Timing Generator to default |
| 273 | settings. |
| 274 | Inputs: none |
| 275 | Outputs: function returns SUCCESS or FAILURE |
| 276 | */ |
| 277 | int tmg_init() |
| 278 | { |
| 279 | if(Reset) |
| 280 | { |
| 281 | return(SUCCESS); |
| 282 | } |
| 283 | calib.CalCompleted = 0; |
| 284 | return diag_tmg_reset(TRUE); |
| 285 | } |
| 286 | |
| 287 | int tmg_walk1(p) |
| 288 | unsigned char *p; |
| 289 | { |
| 290 | int i, j, returncode; |
| 291 | unsigned char tmp; |
| 292 | returncode = SUCCESS; |
| 293 | for(i = 1; j = 0; j < 8; j++, i <= 1) |
| 294 | { |
| 295 | *p = i; |
| 296 | if((tmp = *p) != i) |
| 297 | { |
| 298 | lm_error("walking1: address %08x wrote %02x read %02x\n", p, i, tmp); |
| 299 | returncode = FAILURE; |
| 300 | } |
| 301 | } |
| 302 | return(returncode); |
| 303 | } |
| 304 | |
| 305 | int tmg_walk0(p) |
| 306 | unsigned char *p; |
| 307 | { |
| 308 | unsigned int j, returncode; |
| 309 | unsigned char i, tmp; |
| 310 | returncode = SUCCESS; |
| 311 | for(i = 1; j = 0; j < 8; i <= 1, j++) |
| 312 | { |
| 313 | *p = ~i; |
| 314 | if((tmp = *p) != ~i) |
| 315 | { |
| 316 | lm_error("walking0: address %08x wrote %02x read %02x\n", p, i, tmp); |
| 317 | returncode = FAILURE; |
| 318 | } |
| 319 | } |
| 320 | return(returncode); |
| 321 | } |
| 322 | |
| 323 | int tmg_reg0walk1(p) |
| 324 | unsigned char *p; |
| 325 | { |
| 326 | int i, j, returncode; |
| 327 | unsigned char tmp; |
| 328 | returncode = SUCCESS; |
| 329 | for(i = 1; j = 0; j < 8; j++, i <= 1) |
| 330 | { |
| 331 | if(j < 2) |
| 332 | { |
| 333 | *p = 0x10; /* got to remove ORSHIFT */ |
| 334 | *p = i 0x10; |
| 335 | if((tmp = *p) != (i 0x10)) |
| 336 | { |
| 337 | lm_error("walking1: address %08x wrote %02x read %02x\n", p, i, tmp); |
| 338 | returncode = FAILURE; |
| 339 | } |
| 340 | } |
| 341 | else |
| 342 | { |
| 343 | *p = i; |
| 344 | if((tmp = *p) != i) |
| 345 | { |
| 346 | lm_error("walking1: address %08x wrote %02x read %02x\n", p, i, tmp); |
| 347 | returncode = FAILURE; |
| 348 | } |
| 349 | } |
| 350 | } |
| 351 | return(returncode); |
| 352 | } |
| 353 | |
| 354 | int tmg_reg0walk0(p) |
| 355 | { |
| 356 | int i, j, returncode; |
| 357 | unsigned char tmp; |
| 358 | returncode = SUCCESS; |
| 359 | for(i = 1; j = 0; j < 8; i <= 1, j++) |
| 360 | { |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_util.c

DATE 5/23/89

PAGE #

TIME 4:41:37 pm

4/212

```

LINE # SOURCE TEXT
361 unsigned char *p;
362 {
363     int j, returncode;
364     unsigned char i, tmp;
365
366     returncode = SUCCESS;
367     *p = 0x0ff; /* remove OBRESET */
368     for(i = 1, j = 0; j < 8; i <= 1, j++)
369     {
370         *p = ~i;
371         if(j != 4)
372         {
373             if((tmp = *p) != ~i)
374             {
375                 lm_error("walking0: address 408x wrote 402x read 402x\n", p, i, tmp);
376                 returncode = FAILURE;
377             }
378         }
379         else
380         {
381             if((tmp = *p) != (~i & 0x0fc))
382             {
383                 lm_error("walking0: address 408x wrote 402x read 402x\n", p, i, tmp);
384                 returncode = FAILURE;
385             }
386             *p = 0x0ff; /* remove OBRESET */
387         }
388     }
389     return(returncode);
390 }
391
392 int tmg_reg2walk0(p)
393 unsigned char *p;
394 {
395     int j, returncode;
396     unsigned char i, tmp;
397
398     returncode = SUCCESS;
399     for(i = 0x7e, j = 0; j < 8; j++)
400     {
401         *p = i;
402         if((tmp = *p) != i)
403         {
404             lm_error("walking0: address 408x wrote 402x read 402x\n", p, i, tmp);
405             returncode = FAILURE;
406         }
407         i = ((i < 1) | 0x01) & 0x7f; /* have to keep values
408                                     /* within proper range */
409     }
410     return(returncode);
411 }
412
413 /*
414  * int checkrate(int, int, unsigned long, unsigned long *)
415  * This routine checks rate of the clock using the 8254
416  * Input: value of N
417  * value of output division (2,4,6,...,512,516,520,...,1024)
418  * number of clocks expected
419  * Returns: discrepancy is assigned, function returns SUCCESS
420  * or FAILURE
421  */
422 int tmg_checkrate(n,k,soclocks, discrepancyptr)
423 int n, k;
424 unsigned long soclocks;
425 int *discrepancyptr;
426 {
427     int i, ctrl[3], ctrl2[3], returncode;
428     unsigned long actual, most, least, start;
429
430     if(tmg_clockoff() != SUCCESS) /* shut off clock */
431     {
432         lm_error("Checkrate: cannot turn clock off.\n");
433         return(FAILURE);
434     }
435     returncode = SUCCESS;
436     if((n < 128) || (n > 256))
437     {
438         lm_warning("Checkrate: 4d invalid value of N passed.\n", n);
439         return(FAILURE);
440     }
441     if((k < 2) || (k > 1024) || (k & 1) || ((k > 512) && (k%4 != 0)))
442     {
443         lm_warning("Checkrate: 4d invalid value of K passed.\n", k);
444         return(FAILURE);
445     }
446     tmgptr->pll_rate = (256 - n) + 1;
447     if(k == 2)
448     {
449         tmgptr->pll_divisor = 255; /* have to for fo/2 */
450         tmgptr->clock_select = 0; /* select fo/2 */
451     }
452     else if(k <= 512)
453     {
454         tmgptr->pll_divisor = (256 - (k >> 1)) + 1; /* fo/2k */
455         tmgptr->clock_select = 1; /* select fo/2k */
456     }
457     else
458     {
459         tmgptr->pll_divisor = (256 - (k >> 2)) + 1; /* fo/4k */
460         tmgptr->clock_select = 2; /* select fo/4k */
461     }
462     lm_delay(6); /* wait a long time for lock */
463     /* now that PLL is ready, do the thing with the 8254 */
464     tmgptr->ctcr_register[3].value = CTRL1CM; /* setup 1,2 first */
465     tmgptr->ctcr_register[1].value = zero;
466     tmgptr->ctcr_register[1].value = zero;
467     tmgptr->ctcr_register[3].value = CTRL2CM;
468     tmgptr->ctcr_register[2].value = zero;
469     tmgptr->ctcr_register[2].value = zero;
470 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
diags/tmg_util.c

DATE 5/23/89
TIME 4:41:37 pm

PAGE #
5/213

```

LINE # SOURCE TEXT
481 tmgptr->ctc_register[3].value = CNTROLATCH, /* latch count */
482 tmgptr->ctc_register[0].value = 0x00, /* msb of 1000 */
483 tmgptr->ctc_register[0].value = 0x01, /* msb of 1000 */
484
485 tmgptr->ctc_intr_clearl = 0; /* make sure things are off */
486 if(tmg_clockon() != SUCCESS)
487 {
488     lm_error("Checkrate: cannot turn clock on.\n");
489     return(FAILURE);
490 }
491 tmgptr->ctc_intr_clearl = 1; /* let her rip */
492
493 start = lm_time();
494 while(!tmgptr->ctc_intr) /* wait for finish */
495     if((lm_time() - start) > 150) /* longest time is 137 ms */
496     {
497         lm_error("Checkrate: timer timed out, a = td, k = td, noclocks = td\n", a, k, noclocks);
498         return(FAILURE);
499     }
500
501 tmgptr->ctc_register[3].value = CNTROLATCH, /* latch count */
502 /* and status */
503 for(i = 0; i < 3; i++)
504     ctrl1[i] = tmgptr->ctc_register[1].value & 0x0ff,
505 for(i = 0; i < 3; i++)
506     ctrl2[i] = tmgptr->ctc_register[2].value & 0x0ff,
507
508 /* compute actual number of clocks that occurred */
509 if(ctrl1[0] & 0x40) /* check null flag */
510     least = 0;
511 else
512 {
513     least = ctrl1[1] + (ctrl1[2] << 8);
514     if(least == 0)
515         least = 1;
516     else
517         least = 0x100011 - (unsigned long)least;
518 }
519
520 if(ctrl2[0] & 0x40) /* check null flag */
521     most = 0;
522 else
523 {
524     most = ctrl2[1] + (ctrl2[2] << 8);
525     if(most == 0)
526         most = 1;
527     else
528         most = 0x100011 - (unsigned long)most;
529 }
530
531 actual = (most << 16) + least;
532
533 *discrepancyptr = actual - noclocks;
534
535 return(returncode);
536 }
537
538 /*
539 tmg_display_error()
540 *
541 * This function displays the error latches for the various
542 * lanes.
543 *
544 * Inputs: lanes to display, bit 0 = lane A, bit 3 = lane D
545 * Returns: lm_error returncode
546 */
547 int tmg_display_error(lanes)
548 int lanes;
549 {
550     int i;
551
552     if(!(lanes & 0x0f)) /* no lanes specified */
553     {
554         return(lm_error("tmg_display_error: no lanes specified\n"));
555     }
556     if(!tmgptr->tmg_intr)
557     {
558         return(lm_error("tmg_display_error: no error condition present\n"));
559     }
560
561     lm_message("\nTMC Error Register Display:\n");
562     lm_message("\nlane(a)\t");
563     for(i = 1; i != 0x10; i <= 1)
564     {
565         if(i & lanes)
566             switch(i)
567             {
568                 case 1: lm_message("\tA"); break;
569                 case 2: lm_message("\tB"); break;
570                 case 4: lm_message("\tC"); break;
571                 case 8: lm_message("\tD"); break;
572                 default: break;
573             }
574     }
575     lm_message("\n\npel_ctrl(2..0)");
576     for(i = 1; i != 0x10; i <= 1)
577     {
578         if(i & lanes)
579             switch(i)
580             {
581                 case 1: lm_message("\ttd", tmgptr->lane_a_pel_control); break;
582                 case 2: lm_message("\ttd", tmgptr->lane_b_pel_control); break;
583                 case 4: lm_message("\ttd", tmgptr->lane_c_pel_control); break;
584                 case 8: lm_message("\ttd", tmgptr->lane_d_pel_control); break;
585                 default: break;
586             }
587     }
588     lm_message("\n\nData Valid");
589     for(i = 1; i != 0x10; i <= 1)
590     {
591         if(i & lanes)
592             switch(i)
593             {
594                 case 1: lm_message("\ttd", tmgptr->lane_a_data_valid); break;
595                 case 2: lm_message("\ttd", tmgptr->lane_b_data_valid); break;
596                 case 4: lm_message("\ttd", tmgptr->lane_c_data_valid); break;
597                 case 8: lm_message("\ttd", tmgptr->lane_d_data_valid); break;
598                 default: break;
599             }
600     }
601     lm_message("\n\n");
602     return(lm_error("Failures in table.\n"));
603 }
604 /*
605 tmg_verify_pel_ctrl_error()

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_util.c

DATE

5/23/89

PAGE #

TIME

4:41:37 pm

6/214

```

LINE #          SOURCE TEXT
601  * This function verifies that the specified bit in the specified lane
602  * disagrees with all other specified lanes.
603  *
604  * Inputs: lanes that participated in the presentation, lane with
605  *         the error, and its bit position
606  *
607  * Outputs: function returns SUCCESS if error is verified, else FAILURE
608  */
609  int tmg_verify_pel_ctrl_error(lanes, badlane, bitpos)
610  int lanes, badlane, bitpos;
611  {
612      int a, b, i, mask, returncode;
613      char lane;
614
615      if(!(lanes & 0x0f))
616      {
617          lm_message("tmg_verify_pel_ctrl: no lanes specified\n");
618          return(SUCCESS);
619      }
620      if(lanes == badlane)
621      {
622          lm_message("tmg_verify_pel_ctrl: only one lane... no discrepancies\n");
623          return(SUCCESS);
624      }
625      if((badlane < 0) || (badlane > 0x0f))
626      {
627          lm_error("tmg_verify_pel_ctrl: invalid bad lane specification\n");
628          return(FAILURE);
629      }
630      if((bitpos > 2) || (bitpos < 0))
631      {
632          lm_error("tmg_verify_pel_ctrl: invalid bit position specified\n");
633          return(FAILURE);
634      }
635      mask = 1 << bitpos;
636
637      returncode = SUCCESS;
638      switch(badlane)
639      {
640          case 1: b = tmgptr->lane_a_pel_control; break;
641          case 2: b = tmgptr->lane_b_pel_control; break;
642          case 4: b = tmgptr->lane_c_pel_control; break;
643          case 8: b = tmgptr->lane_d_pel_control; break;
644          default: break;
645      }
646      for(i = 1; i != 0x10; i <= 1)
647      {
648          if(i == badlane)
649              continue;
650          if(i & lanes)
651          {
652              switch(i)
653              {
654                  case 1: a = tmgptr->lane_a_pel_control; lane = 'A'; break;
655                  case 2: a = tmgptr->lane_b_pel_control; lane = 'B'; break;
656                  case 4: a = tmgptr->lane_c_pel_control; lane = 'C'; break;
657                  case 8: a = tmgptr->lane_d_pel_control; lane = 'D'; break;
658                  default: break;
659              }
660              if(((a ^ b) & mask))
661              {
662                  lm_error("tmg_verify_pel_ctrl: bad data from lane %c, bit %d\n", lane, bitpos);
663                  returncode = FAILURE;
664              }
665          }
666      }
667      return(returncode);
668  }
669
670  /* int tmg_verify_data_valid_error()
671  *
672  * This function verifies that the specified data valid bit in the
673  * specified lane disagrees with all other specified lanes.
674  *
675  * Inputs: lanes participating in the presentation, lane to check
676  *
677  * Outputs: function returns SUCCESS if error is verified, else FAILURE
678  */
679  int tmg_verify_data_valid_error(lanes, badlane)
680  int lanes, badlane;
681  {
682      int a, b, i, returncode;
683      char lane;
684
685      returncode = SUCCESS;
686
687      if(!(lanes & 0x0f))
688      {
689          lm_message("tmg_verify_data_valid_error: no lanes specified\n");
690          return(SUCCESS);
691      }
692      if(lanes == badlane)
693      {
694          lm_message("tmg_verify_data_valid_error: only one lane, no discrepancy\n");
695          return(SUCCESS);
696      }
697      if((badlane < 0) || (badlane > 0x0f))
698      {
699          lm_error("tmg_verify_data_valid_error: invalid bad lane specified\n");
700          return(FAILURE);
701      }
702      switch(badlane)
703      {
704          case 1: b = tmgptr->lane_a_data_valid; break;
705          case 2: b = tmgptr->lane_b_data_valid; break;
706          case 4: b = tmgptr->lane_c_data_valid; break;
707          case 8: b = tmgptr->lane_d_data_valid; break;
708          default: break;
709      }
710      for(i = 1; i != 0x10; i <= 1)
711      {
712          if(i == badlane)
713              continue;
714          if(i & lanes)
715          {
716              switch(i)
717              {
718                  case 1: a = tmgptr->lane_a_data_valid; lane = 'A'; break;
719                  case 2: a = tmgptr->lane_b_data_valid; lane = 'B'; break;
720                  case 4: a = tmgptr->lane_c_data_valid; lane = 'C'; break;
721                  case 8: a = tmgptr->lane_d_data_valid; lane = 'D'; break;

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_util.c

DATE

5/23/89

PAGE #

TIME

4:41:37 pm

7/215

```

721      default: break;
722      }
723      if((a == b))
724      {
725          tmg_verify_data_valid_error: no disagreement between bad lane and lane to(a", lane);
726          returncode = FAILURE;
727      }
728      }
729      return(returncode);
730  }
731  }
732
733  /*
734  int tmg_clear_error(void)
735  *
736  * This function clears the TMG detected backplane error.
737  * It returns SUCCESS if successful, else ERROR.
738  */
739  int tmg_clear_error()
740  {
741      if(!tmgptr->tmg_intr) /* no interrupt to clear */
742          return(SUCCESS);
743      tmgptr->tmg_intr_clearl = 0;
744      tmgptr->tmg_intr_clearh = 1;
745      if(!tmgptr->clock_on)
746      {
747          if(tmg_clockon() != SUCCESS)
748          {
749              tmg_report_failure(mg, "tmg_play");
750              return(FAILURE);
751          }
752          if(tmg_clockoff() != SUCCESS)
753          {
754              tmg_report_failure(mg, "tmg_play");
755              return(FAILURE);
756          }
757      }
758      return(SUCCESS);
759  }
760
761  /* the following functions were previously in tmg_cal.c */
762  tmg_play_til_key()
763  {
764      static char msg[] = "tmg_play_til_key";
765      int message("PRESS ANY KEY TO CONTINUE...\n");
766      do {
767          if(tmg_play(TIMEOUT) != SUCCESS) {
768              tmg_report_failure(mg, "tmg_play");
769              return(FAILURE);
770          }
771          while (tmg_check_key() == 0);
772          int message("PRESS ANY KEY TO CONTINUE...\n");
773          while(tmg_check_key() != 0) tmg_get_key();
774          return SUCCESS;
775      }
776  }
777
778  long tmg_plot_delays()
779  {
780      INTR_INIT
781      register long ramp, edge;
782      long minramp = 0;
783      long minedge = 0;
784      long maxramp = NUMBER_OF_ERAMPS - 1;
785      long maxedge = NUMBER_OF_EDGES + 1;
786      long decrement = 0;
787
788      diag_get_long(minramp, "start ramp", 0, maxramp);
789      diag_get_long(maxramp, "end ramp", minramp, maxramp);
790      diag_get_long(minedge, "start edge", 0, maxedge);
791      diag_get_long(maxedge, "end edge", minedge, maxedge);
792      diag_get_long(decrement, "decrement = 1, increment = 0", 0, 1);
793
794      INTR_BEGIN
795      tmg_set_test_mode(1); tmg_set_slot_count(1);
796      for (ramp = minramp; ramp <= maxramp; ++ramp) {
797          for (edge = minedge; edge <= maxedge; ++edge) {
798              tmg_plot_delay(ramp, edge, decrement);
799          }
800      }
801      INTR_END
802      tmg_restore_calibration();
803      intr();
804      tmg_restore_calibration();
805      return SUCCESS;
806  }
807
808  tmg_plot_delay(ramp, edge, decrement)
809  register u_long ramp, edge;
810  {
811      static char msg[] = "tmg_plot_delay";
812      long p1, p2, c1, c2, junk;
813      register long thr, minth;
814      int rthr, stopthr, incthr;
815
816      tmg_set_slot_count(2);
817
818      switch (edge) {
819      default:
820          thr = tmg_predict_threshold(40000);
821          calib.EdgeSlope[ramp][edge], calib.EdgeOffset[ramp][edge];
822          minth = calib.EdgeMinThresh[ramp][edge];
823          break;
824      case 6: /* Edge 7 */
825          thr = (40000 - calib.SampleOffset[ramp][0]) / calib.Edge7Slope[ramp];
826          minth = calib.Edge7MinThresh[ramp];
827          tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE;
828          break;
829      case 7: /* Sample Ramp */
830          thr = tmg_predict_threshold(40000);
831          calib.SampleSlope[ramp], calib.EarlySampleOffset[ramp];
832          minth = calib.SampleMinThresh[ramp];
833          tmgptr->sample_mode = EARLYSAMPLETRIGGERMODE;
834      }
835  }

```


Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg_util.c

DATE

5/23/89

PAGE #

TIME

4:41:37 pm

8/216

LINE # SOURCE TEXT

```

841     break;
842 }
843 if (thr < minth)
844     thr = minth;
845
846 if (decrement) {
847     startthr = 250;
848     stopthr = thr - 1;
849     incthr = -1;
850 } else {
851     startthr = thr;
852     stopthr = 251;
853     incthr = 1;
854 }
855
856 for (thr = startthr; thr != stopthr; thr += incthr) {
857     switch (edge) {
858     default:
859         if (! (p1 = tmg_find_period(ramp, edge, thr))) {
860             tmg_report_failure(me, "tmg_find_period");
861             return(FAILURE);
862         }
863         break;
864     case 6: /* Edge 7 */
865         if (! (p1 = tmg_find_sample_period(ramp, 0, thr, 10))) {
866             tmg_report_failure(me, "tmg_find_sample_period");
867             return(FAILURE);
868         }
869         break;
870     case 7: /* Sample Ramp */
871         if (! (p1 = tmg_find_sample_period(3, ramp, 255, thr))) {
872             tmg_report_failure(me, "tmg_find_sample_period");
873             return(FAILURE);
874         }
875         break;
876     }
877     if (tmg_get_next_lower_period(p1,
878         &p2, &junk, &junk, &junk, &junk) != SUCCESS) {
879         tmg_report_failure(me, "tmg_get_frequency_setting");
880         return FAILURE;
881     }
882     switch (edge) {
883     default:
884         o1 = tmg_predict_offset(p1, thr, calib.Edgeslope[ramp][edge]);
885         o2 = tmg_predict_offset(p2, thr, calib.Edgeslope[ramp][edge]);
886         break;
887     case 6: /* Edge 7 */
888         o1 = p1 - calib.Edge7Slope[ramp] * thr - calib.SampleSlope[0] * 10;
889         o2 = p2 - calib.Edge7Slope[ramp] * thr - calib.SampleSlope[0] * 10;
890         break;
891     case 7: /* Sample Ramp */
892         o1 = tmg_predict_offset(p1, thr, calib.SampleSlope[ramp]);
893         o2 = tmg_predict_offset(p2, thr, calib.SampleSlope[ramp]);
894         break;
895     }
896     lm_message("td td tld: t10d t10d t10d t10d\n", ramp, edge, thr,
897         p1, o1, p2, o2);
898 }
899
900 tmg_set_slot_count(1);
901 return SUCCESS;
902

```

| | | | | |
|--|--|-------------------------------|---------------------------------------|---------------|
| Copyright 1989 Logic Modeling Systems | | FILE bsp.diags/bsp.assem.s | DATE 5/23/89 TIME 4:41:07 pm | PAGE # 1/1 |
|--|--|-------------------------------|---------------------------------------|---------------|

| | |
|--------|---|
| LINE # | TEXT |
| 1 | SCCS_ID: bsp.assem.s rev 3.1, 4/24/89 at 07:54:58 |
| 2 | ----- File Defines ----- |
| 3 | |
| 4 | |
| 5 | ===== |
| 6 | DEFINITIONS SPECIFIC FOR THE CPU BOARD |
| 7 | ===== |
| 8 | |
| 9 | |
| 10 | |
| 11 | DISINT = 0x3700 |
| 12 | EMAIPT = 0x3000 |
| 13 | RTSCOPE = 0x01 |
| 14 | VRTX = 0x00 |
| 15 | IVT_BASE = 0x000000 |
| 16 | DUART_SR = 0x10c10014 |
| 17 | STATUS_REG_A = 0x10c10004 |
| 18 | STATUS_REG_B = 0x10c10014 |
| 19 | DUART_RX_TX_REG_A = 0x10c1000c |
| 20 | DUART_RX_TX_REG_B = 0x10c1002c |
| 21 | CHAR_TX_A = 24 |
| 22 | CHAR_RX_A = 25 |
| 23 | CHAR_TX_B = 28 |
| 24 | CHAR_RX_B = 29 |
| 25 | CPU_INTR_REG = 0x10c60001 |
| 26 | TIMER2_CONTROL = 0x10c1000c |
| 27 | TIMER2_COUNT = 0x10c10008 |
| 28 | READ_COUNTER2 = 0x00000000 |
| 29 | TIMER2_TOTAL = 0x013E |
| 30 | TIMER2_LOW_COUNT = 0x40000000 |
| 31 | TIMER2_HIGH_COUNT = 0x10000000 |
| 32 | TIMER2_LOW_COUNT = 0x20000000 |
| 33 | TIMER2_HIGH_COUNT = 0x00000000 |
| 34 | ENABLE_LANCE = 0x00 |
| 35 | CPU_PMC_REG = 0x10c60001 |
| 36 | |
| 37 | |
| 38 | VRTX and RTSCOPE function codes |
| 39 | ===== |
| 40 | |
| 41 | |
| 42 | TR_ENTER = 0x0102 |
| 43 | TR_EXIT = 0x0103 |
| 44 | TR_ENTER = 0x0104 |
| 45 | TR_EXIT = 0x0105 |
| 46 | UI_TIMER = 0x0012 |
| 47 | UI_TIMER = 0x0013 |
| 48 | UI_TIMER = 0x0014 |
| 49 | UI_TIMER = 0x0015 |
| 50 | UI_TIMER = 0x0016 |
| 51 | UI_TIMER = 0x0017 |
| 52 | UI_TIMER = 0x0018 |
| 53 | UI_TIMER = 0x0019 |
| 54 | UI_TIMER = 0x001A |
| 55 | UI_TIMER = 0x001B |
| 56 | UI_TIMER = 0x001C |
| 57 | UI_TIMER = 0x001D |
| 58 | UI_TIMER = 0x001E |
| 59 | UI_TIMER = 0x001F |
| 60 | UI_TIMER = 0x0020 |
| 61 | UI_TIMER = 0x0021 |
| 62 | UI_TIMER = 0x0022 |
| 63 | UI_TIMER = 0x0023 |
| 64 | UI_TIMER = 0x0024 |
| 65 | UI_TIMER = 0x0025 |
| 66 | UI_TIMER = 0x0026 |
| 67 | UI_TIMER = 0x0027 |
| 68 | UI_TIMER = 0x0028 |
| 69 | UI_TIMER = 0x0029 |
| 70 | UI_TIMER = 0x002A |
| 71 | UI_TIMER = 0x002B |
| 72 | UI_TIMER = 0x002C |
| 73 | UI_TIMER = 0x002D |
| 74 | UI_TIMER = 0x002E |
| 75 | UI_TIMER = 0x002F |
| 76 | UI_TIMER = 0x0030 |
| 77 | UI_TIMER = 0x0031 |
| 78 | UI_TIMER = 0x0032 |
| 79 | UI_TIMER = 0x0033 |
| 80 | UI_TIMER = 0x0034 |
| 81 | UI_TIMER = 0x0035 |
| 82 | UI_TIMER = 0x0036 |
| 83 | UI_TIMER = 0x0037 |
| 84 | UI_TIMER = 0x0038 |
| 85 | UI_TIMER = 0x0039 |
| 86 | UI_TIMER = 0x003A |
| 87 | UI_TIMER = 0x003B |
| 88 | UI_TIMER = 0x003C |
| 89 | UI_TIMER = 0x003D |
| 90 | UI_TIMER = 0x003E |
| 91 | UI_TIMER = 0x003F |
| 92 | UI_TIMER = 0x0040 |
| 93 | UI_TIMER = 0x0041 |
| 94 | UI_TIMER = 0x0042 |
| 95 | UI_TIMER = 0x0043 |
| 96 | UI_TIMER = 0x0044 |
| 97 | UI_TIMER = 0x0045 |
| 98 | UI_TIMER = 0x0046 |
| 99 | UI_TIMER = 0x0047 |
| 100 | UI_TIMER = 0x0048 |
| 101 | UI_TIMER = 0x0049 |
| 102 | UI_TIMER = 0x004A |
| 103 | UI_TIMER = 0x004B |
| 104 | UI_TIMER = 0x004C |
| 105 | UI_TIMER = 0x004D |
| 106 | UI_TIMER = 0x004E |
| 107 | UI_TIMER = 0x004F |
| 108 | UI_TIMER = 0x0050 |
| 109 | UI_TIMER = 0x0051 |
| 110 | UI_TIMER = 0x0052 |
| 111 | UI_TIMER = 0x0053 |
| 112 | UI_TIMER = 0x0054 |
| 113 | UI_TIMER = 0x0055 |
| 114 | UI_TIMER = 0x0056 |
| 115 | UI_TIMER = 0x0057 |
| 116 | UI_TIMER = 0x0058 |
| 117 | UI_TIMER = 0x0059 |
| 118 | UI_TIMER = 0x005A |
| 119 | UI_TIMER = 0x005B |
| 120 | UI_TIMER = 0x005C |
| 121 | UI_TIMER = 0x005D |
| 122 | UI_TIMER = 0x005E |
| 123 | UI_TIMER = 0x005F |
| 124 | UI_TIMER = 0x0060 |
| 125 | UI_TIMER = 0x0061 |
| 126 | UI_TIMER = 0x0062 |
| 127 | UI_TIMER = 0x0063 |
| 128 | UI_TIMER = 0x0064 |
| 129 | UI_TIMER = 0x0065 |
| 130 | UI_TIMER = 0x0066 |
| 131 | UI_TIMER = 0x0067 |
| 132 | UI_TIMER = 0x0068 |
| 133 | UI_TIMER = 0x0069 |
| 134 | UI_TIMER = 0x006A |
| 135 | UI_TIMER = 0x006B |
| 136 | UI_TIMER = 0x006C |
| 137 | UI_TIMER = 0x006D |
| 138 | UI_TIMER = 0x006E |
| 139 | UI_TIMER = 0x006F |
| 140 | UI_TIMER = 0x0070 |
| 141 | UI_TIMER = 0x0071 |
| 142 | UI_TIMER = 0x0072 |
| 143 | UI_TIMER = 0x0073 |
| 144 | UI_TIMER = 0x0074 |
| 145 | UI_TIMER = 0x0075 |
| 146 | UI_TIMER = 0x0076 |
| 147 | UI_TIMER = 0x0077 |
| 148 | UI_TIMER = 0x0078 |
| 149 | UI_TIMER = 0x0079 |
| 150 | UI_TIMER = 0x007A |
| 151 | UI_TIMER = 0x007B |
| 152 | UI_TIMER = 0x007C |
| 153 | UI_TIMER = 0x007D |
| 154 | UI_TIMER = 0x007E |
| 155 | UI_TIMER = 0x007F |
| 156 | UI_TIMER = 0x0080 |
| 157 | UI_TIMER = 0x0081 |
| 158 | UI_TIMER = 0x0082 |
| 159 | UI_TIMER = 0x0083 |
| 160 | UI_TIMER = 0x0084 |
| 161 | UI_TIMER = 0x0085 |
| 162 | UI_TIMER = 0x0086 |
| 163 | UI_TIMER = 0x0087 |
| 164 | UI_TIMER = 0x0088 |
| 165 | UI_TIMER = 0x0089 |
| 166 | UI_TIMER = 0x008A |
| 167 | UI_TIMER = 0x008B |
| 168 | UI_TIMER = 0x008C |
| 169 | UI_TIMER = 0x008D |
| 170 | UI_TIMER = 0x008E |
| 171 | UI_TIMER = 0x008F |
| 172 | UI_TIMER = 0x0090 |
| 173 | UI_TIMER = 0x0091 |
| 174 | UI_TIMER = 0x0092 |
| 175 | UI_TIMER = 0x0093 |
| 176 | UI_TIMER = 0x0094 |
| 177 | UI_TIMER = 0x0095 |
| 178 | UI_TIMER = 0x0096 |
| 179 | UI_TIMER = 0x0097 |
| 180 | UI_TIMER = 0x0098 |
| 181 | UI_TIMER = 0x0099 |
| 182 | UI_TIMER = 0x009A |
| 183 | UI_TIMER = 0x009B |
| 184 | UI_TIMER = 0x009C |
| 185 | UI_TIMER = 0x009D |
| 186 | UI_TIMER = 0x009E |
| 187 | UI_TIMER = 0x009F |
| 188 | UI_TIMER = 0x00A0 |
| 189 | UI_TIMER = 0x00A1 |
| 190 | UI_TIMER = 0x00A2 |
| 191 | UI_TIMER = 0x00A3 |
| 192 | UI_TIMER = 0x00A4 |
| 193 | UI_TIMER = 0x00A5 |
| 194 | UI_TIMER = 0x00A6 |
| 195 | UI_TIMER = 0x00A7 |
| 196 | UI_TIMER = 0x00A8 |
| 197 | UI_TIMER = 0x00A9 |
| 198 | UI_TIMER = 0x00AA |
| 199 | UI_TIMER = 0x00AB |
| 200 | UI_TIMER = 0x00AC |
| 201 | UI_TIMER = 0x00AD |
| 202 | UI_TIMER = 0x00AE |
| 203 | UI_TIMER = 0x00AF |
| 204 | UI_TIMER = 0x00B0 |
| 205 | UI_TIMER = 0x00B1 |
| 206 | UI_TIMER = 0x00B2 |
| 207 | UI_TIMER = 0x00B3 |
| 208 | UI_TIMER = 0x00B4 |
| 209 | UI_TIMER = 0x00B5 |
| 210 | UI_TIMER = 0x00B6 |

| | |
|-----|---|
| 1 | SCCS_ID: bsp.assem.s rev 3.1, 4/24/89 at 07:54:58 |
| 2 | ----- File Defines ----- |
| 3 | |
| 4 | |
| 5 | ===== |
| 6 | DEFINITIONS SPECIFIC FOR THE CPU BOARD |
| 7 | ===== |
| 8 | |
| 9 | |
| 10 | |
| 11 | DISINT = 0x3700 |
| 12 | EMAIPT = 0x3000 |
| 13 | RTSCOPE = 0x01 |
| 14 | VRTX = 0x00 |
| 15 | IVT_BASE = 0x000000 |
| 16 | DUART_SR = 0x10c10014 |
| 17 | STATUS_REG_A = 0x10c10004 |
| 18 | STATUS_REG_B = 0x10c10014 |
| 19 | DUART_RX_TX_REG_A = 0x10c1000c |
| 20 | DUART_RX_TX_REG_B = 0x10c1002c |
| 21 | CHAR_TX_A = 24 |
| 22 | CHAR_RX_A = 25 |
| 23 | CHAR_TX_B = 28 |
| 24 | CHAR_RX_B = 29 |
| 25 | CPU_INTR_REG = 0x10c60001 |
| 26 | TIMER2_CONTROL = 0x10c1000c |
| 27 | TIMER2_COUNT = 0x10c10008 |
| 28 | READ_COUNTER2 = 0x00000000 |
| 29 | TIMER2_TOTAL = 0x013E |
| 30 | TIMER2_LOW_COUNT = 0x40000000 |
| 31 | TIMER2_HIGH_COUNT = 0x10000000 |
| 32 | TIMER2_LOW_COUNT = 0x20000000 |
| 33 | TIMER2_HIGH_COUNT = 0x00000000 |
| 34 | ENABLE_LANCE = 0x00 |
| 35 | CPU_PMC_REG = 0x10c60001 |
| 36 | |
| 37 | |
| 38 | VRTX and RTSCOPE function codes |
| 39 | ===== |
| 40 | |
| 41 | |
| 42 | TR_ENTER = 0x0102 |
| 43 | TR_EXIT = 0x0103 |
| 44 | TR_ENTER = 0x0104 |
| 45 | TR_EXIT = 0x0105 |
| 46 | UI_TIMER = 0x0012 |
| 47 | UI_TIMER = 0x0013 |
| 48 | UI_TIMER = 0x0014 |
| 49 | UI_TIMER = 0x0015 |
| 50 | UI_TIMER = 0x0016 |
| 51 | UI_TIMER = 0x0017 |
| 52 | UI_TIMER = 0x0018 |
| 53 | UI_TIMER = 0x0019 |
| 54 | UI_TIMER = 0x001A |
| 55 | UI_TIMER = 0x001B |
| 56 | UI_TIMER = 0x001C |
| 57 | UI_TIMER = 0x001D |
| 58 | UI_TIMER = 0x001E |
| 59 | UI_TIMER = 0x001F |
| 60 | UI_TIMER = 0x0020 |
| 61 | UI_TIMER = 0x0021 |
| 62 | UI_TIMER = 0x0022 |
| 63 | UI_TIMER = 0x0023 |
| 64 | UI_TIMER = 0x0024 |
| 65 | UI_TIMER = 0x0025 |
| 66 | UI_TIMER = 0x0026 |
| 67 | UI_TIMER = 0x0027 |
| 68 | UI_TIMER = 0x0028 |
| 69 | UI_TIMER = 0x0029 |
| 70 | UI_TIMER = 0x002A |
| 71 | UI_TIMER = 0x002B |
| 72 | UI_TIMER = 0x002C |
| 73 | UI_TIMER = 0x002D |
| 74 | UI_TIMER = 0x002E |
| 75 | UI_TIMER = 0x002F |
| 76 | UI_TIMER = 0x0030 |
| 77 | UI_TIMER = 0x0031 |
| 78 | UI_TIMER = 0x0032 |
| 79 | UI_TIMER = 0x0033 |
| 80 | UI_TIMER = 0x0034 |
| 81 | UI_TIMER = 0x0035 |
| 82 | UI_TIMER = 0x0036 |
| 83 | UI_TIMER = 0x0037 |
| 84 | UI_TIMER = 0x0038 |
| 85 | UI_TIMER = 0x0039 |
| 86 | UI_TIMER = 0x003A |
| 87 | UI_TIMER = 0x003B |
| 88 | UI_TIMER = 0x003C |
| 89 | UI_TIMER = 0x003D |
| 90 | UI_TIMER = 0x003E |
| 91 | UI_TIMER = 0x003F |
| 92 | UI_TIMER = 0x0040 |
| 93 | UI_TIMER = 0x0041 |
| 94 | UI_TIMER = 0x0042 |
| 95 | UI_TIMER = 0x0043 |
| 96 | UI_TIMER = 0x0044 |
| 97 | UI_TIMER = 0x0045 |
| 98 | UI_TIMER = 0x0046 |
| 99 | UI_TIMER = 0x0047 |
| 100 | UI_TIMER = 0x0048 |
| 101 | UI_TIMER = 0x0049 |
| 102 | UI_TIMER = 0x004A |
| 103 | UI_TIMER = 0x004B |
| 104 | UI_TIMER = 0x004C |
| 105 | UI_TIMER = 0x004D |
| 106 | UI_TIMER = 0x004E |
| 107 | UI_TIMER = 0x004F |
| 108 | UI_TIMER = 0x0050 |
| 109 | UI_TIMER = 0x0051 |
| 110 | UI_TIMER = 0x0052 |
| 111 | UI_TIMER = 0x0053 |
| 112 | UI_TIMER = 0x0054 |
| 113 | UI_TIMER = 0x0055 |
| 114 | UI_TIMER = 0x0056 |
| 115 | UI_TIMER = 0x0057 |
| 116 | UI_TIMER = 0x0058 |
| 117 | UI_TIMER = 0x0059 |
| 118 | UI_TIMER = 0x005A |
| 119 | UI_TIMER = 0x005B |
| 120 | UI_TIMER = 0x005C |
| 121 | UI_TIMER = 0x005D |
| 122 | UI_TIMER = 0x005E |
| 123 | UI_TIMER = 0x005F |
| 124 | UI_TIMER = 0x0060 |
| 125 | UI_TIMER = 0x0061 |
| 126 | UI_TIMER = 0x0062 |
| 127 | UI_TIMER = 0x0063 |
| 128 | UI_TIMER = 0x0064 |
| 129 | UI_TIMER = 0x0065 |
| 130 | UI_TIMER = 0x0066 |
| 131 | UI_TIMER = 0x0067 |
| 132 | UI_TIMER = 0x0068 |
| 133 | UI_TIMER = 0x0069 |
| 134 | UI_TIMER = 0x006A |
| 135 | UI_TIMER = 0x006B |
| 136 | UI_TIMER = 0x006C |
| 137 | UI_TIMER = 0x006D |
| 138 | UI_TIMER = 0x006E |
| 139 | UI_TIMER = 0x006F |
| 140 | UI_TIMER = 0x0070 |
| 141 | UI_TIMER = 0x0071 |
| 142 | UI_TIMER = 0x0072 |
| 143 | UI_TIMER = 0x0073 |
| 144 | UI_TIMER = 0x0074 |
| 145 | UI_TIMER = 0x0075 |
| 146 | UI_TIMER = 0x0076 |
| 147 | UI_TIMER = 0x0077 |
| 148 | UI_TIMER = 0x0078 |
| 149 | UI_TIMER = 0x0079 |
| 150 | UI_TIMER = 0x007A |
| 151 | UI_TIMER = 0x007B |
| 152 | UI_TIMER = 0x007C |
| 153 | UI_TIMER = 0x007D |
| 154 | UI_TIMER = 0x007E |
| 155 | UI_TIMER = 0x007F |
| 156 | UI_TIMER = 0x0080 |
| 157 | UI_TIMER = 0x0081 |
| 158 | UI_TIMER = 0x0082 |
| 159 | UI_TIMER = 0x0083 |
| 160 | UI_TIMER = 0x0084 |
| 161 | UI_TIMER = 0x0085 |
| 162 | UI_TIMER = 0x0086 |
| 163 | UI_TIMER = 0x0087 |
| 164 | UI_TIMER = 0x0088 |
| 165 | UI_TIMER = 0x0089 |
| 166 | UI_TIMER = 0x008A |
| 167 | UI_TIMER = 0x008B |
| 168 | UI_TIMER = 0x008C |
| 169 | UI_TIMER = 0x008D |
| 170 | UI_TIMER = 0x008E |
| 171 | UI_TIMER = 0x008F |
| 172 | UI_TIMER = 0x0090 |
| 173 | UI_TIMER = 0x0091 |
| 174 | UI_TIMER = 0x0092 |
| 175 | UI_TIMER = 0x0093 |
| 176 | UI_TIMER = 0x0094 |
| 177 | UI_TIMER = 0x0095 |
| 178 | UI_TIMER = 0x0096 |
| 179 | UI_TIMER = 0x0097 |
| 180 | UI_TIMER = 0x0098 |
| 181 | UI_TIMER = 0x0099 |
| 182 | UI_TIMER = 0x009A |
| 183 | UI_TIMER = 0x009B |
| 184 | UI_TIMER = 0x009C |
| 185 | UI_TIMER = 0x009D |
| 186 | UI_TIMER = 0x009E |
| 187 | UI_TIMER = 0x009F |
| 188 | UI_TIMER = 0x00A0 |
| 189 | UI_TIMER = 0x00A1 |
| 190 | UI_TIMER = 0x00A2 |
| 191 | UI_TIMER = 0x00A3 |
| 192 | UI_TIMER = 0x00A4 |
| 193 | UI_TIMER = 0x00A5 |
| 194 | UI_TIMER = 0x00A6 |
| 195 | UI_TIMER = 0x00A7 |
| 196 | UI_TIMER = 0x00A8 |
| 197 | UI_TIMER = 0x00A9 |
| 198 | UI_TIMER = 0x00AA |
| 199 | UI_TIMER = 0x00AB |
| 200 | UI_TIMER = 0x00AC |
| 201 | UI_TIMER = 0x00AD |
| 202 | UI_TIMER = 0x00AE |
| 203 | UI_TIMER = 0x00AF |
| 204 | UI_TIMER = 0x00B0 |
| 205 | UI_TIMER = 0x00B1 |
| 206 | UI_TIMER = 0x00B2 |
| 207 | UI_TIMER = 0x00B3 |
| 208 | UI_TIMER = 0x00B4 |
| 209 | UI_TIMER = 0x00B5 |
| 210 | UI_TIMER = 0x00B6 |

| | |
|----|---|
| 1 | SCCS_ID: bsp.assem.s rev 3.1, 4/24/89 at 07:54:58 |
| 2 | ----- File Defines ----- |
| 3 | |
| 4 | |
| 5 | ===== |
| 6 | DEFINITIONS SPECIFIC FOR THE CPU BOARD |
| 7 | ===== |
| 8 | |
| 9 | |
| 10 | |
| 11 | DISINT = 0x3700 |
| 12 | EMAIPT = 0x3000 |
| 13 | RTSCOPE = 0x01 |
| 14 | VRTX = 0x00 |
| 15 | IVT_BASE = 0x000000 |
| 16 | DUART_SR = 0x10c10014 |
| 17 | STATUS_REG_A = 0x10c10004 |
| 18 | STATUS_REG_B = 0x10c10014 |
| 19 | DUART_RX_TX_REG_A = 0x10c1000c |
| 20 | DUART_RX_TX_REG_B = 0x10c1002c |
| 21 | CHAR_TX_A = 24 |
| 22 | CHAR_RX_A = 25 |
| 23 | CHAR_TX_B = 28 |
| 24 | CHAR_RX_B = 29 |
| 25 | CPU_INTR_REG = 0x10c60001 |
| 26 | TIMER2_CONTROL = 0x10c1000c |
| 27 | TIMER2_COUNT = 0x10c10008 |
| 28 | READ_COUNTER2 = 0x00000000 |
| 29 | TIMER2_TOTAL = 0x013E |
| 30 | TIMER2_LOW_COUNT = 0x40000000 |
| 31 | TIMER2_HIGH_COUNT = 0x10000000 |
| 32 | TIMER2_LOW_COUNT = 0x20000000 |
| 33 | TIMER2_HIGH_COUNT = 0x00000000 |
| 34 | ENABLE_LANCE = 0x00 |
| 35 | CPU_PMC_REG = 0x10c60001 |
| 36 | |
| 37 | |
| 38 | VRTX and RTSCOPE function codes |
| 39 | ===== |
| 40 | |
| 41 | |
| 42 | TR_ENTER = 0x0102 |
| 43 | TR_EXIT = 0x0103 |
| 44 | TR_ENTER = 0x0104 |
| 45 | TR_EXIT = 0x0105 |
| 46 | UI_TIMER = 0x0012 |
| 47 | UI_TIMER = 0x0013 |
| 48 | UI_TIMER = 0x0014 |
| 49 | UI_TIMER = 0x0015 |
| 50 | UI_TIMER = 0x0016 |
| | |

Copyright 1989

Logic Modeling Systems

FILE

bsp.diags/bsp.assem.s

DATE

5/23/89

PAGE #

TIME

4:41:07 pm

2/2

```

LINE #      TEXT
121      .globl _vtx_init
122      _vtx_init:
123      movl $VCTX_INIT,d0      Load Vctx Init Function code
124      trap $VCTX              Trap to Vctx
125      rts                     Return Error to C is DO
126
127
128
129
130
131      ---- Rtscope Init Interface
132      .globl _rts_init
133      _rts_init:
134      #ifdef RTSCOPE_DIAGS_DEBUG
135      link a6,0
136      movl a6,sp@-             frame pointer
137      movl a6(0),a0            save A0 on the stack
138      movl $RTSCOPE_CONST,a0   RTSCOPE const. table addr
139      trap $RTSCOPE            DSINIT
140      movl sp@+,a0             call RTSCOPE
141      rts                     restore A0
142      #endif
143
144
145
146
147
148
149      ---- Rtscope Go Interface
150      .globl _rts_go
151      _rts_go:
152      #ifdef RTSCOPE_DIAGS_DEBUG
153      movl $RTSCOPE_CONST,d0   DSCO
154      trap $RTSCOPE            call RTSCOPE
155      #endif
156      movl $main,a0            New task address
157      movl $MAIN_THROD,d1      Task mode
158      movb $MAIN_TID,d1        Task ID number
159      movb $MAIN_PRI,d1        Task priority
160      movl $SC_CREATE,d0       System call code
161      trap $VCTX              Call VCTX
162
163      ---- Issue VCTX GO system call
164      movl $SC_GO,d0           SC_GO
165      trap $VCTX              Shouldn't come back
166      rts
167
168
169      bad_start:
170      jmp _bad_start
171
172
173
174
175      =====
176      Interrupt Service Routines
177
178
179      ---- Clock Board Error ISR
180      .globl _error_isr, _parity_isr
181      _error_isr:
182      movl d0,sp@-
183      movl $0x7ffe,sp@-
184      jar _error_isr           dont save a7 & d0 save d1-d7,a0-a6
185      movl sp@+,0x7ffe         This is our C interrupt routine
186      rts                     restore above regs.
187
188      ---- Perform UI_EXIT from ISR through Vctx
189      movl $UI_EXIT,d0
190      trap $VCTX
191
192
193      ---- Parity ISR
194      _parity_isr:
195      movl d0,sp@-
196      movl $0x7ffe,sp@-
197      jar _parity_isr           dont save a7 & d0 save d1-d7,a0-a6
198      movl sp@+,0x7ffe         This is our C interrupt routine
199      rts                     restore above regs.
200
201      ---- Perform UI_EXIT from ISR through Vctx
202      movl $UI_EXIT,d0
203      trap $VCTX
204
205
206      ---- Ethernet ISR
207      .globl _ether_isr
208      _ether_isr:
209      movl d0,sp@-
210      movl $0x7ffe,sp@-
211      movl $0,_system_call
212      jar _ether_isr           dont save a7 & d0 save d1-d7,a0-a6
213      movl _system_call,d0     check if a system call is made?
214      movl sp@+,0x7ffe         This is our C interrupt routine
215      btst $0,d0              let us check bit 0
216      beq no_sys_calls         restore above regs.
217      rts                     should we exit thru VCTX
218      go to no system calls, if no VCTX
219      calls were made.
220
221      ---- Perform UI_EXIT from ISR through Vctx
222      movl $UI_EXIT,d0
223      trap $VCTX
224
225      ---- NO SYSTEM CALLS WERE MADE
226      no_sys_calls:
227      movl sp@+,d0
228      rts
229
230
231      ---- DUART ISR
232      Chassel A Serial Port - RX buffer full
233      .globl _serial_isr
234      _serial_isr:
235      movl d0,sp@-
236      movl d1,sp@-
237      movl a0,sp@-
238      movl $DUART_SR,a0        get status

```

| Copyright 1989 Logic Modeling Systems | | FILE bsp.diags/bsp.assem.s | DATE 5/23/89 TIME 4:41:07 pm | PAGE # 3/3 |
|--|---|---|---------------------------------------|---------------|
| LINE # | TEXT | | | |
| 241 | movl a0,d1 | load into reg | | |
| 242 | andl %uart_mask,d1 | only look @ interrupts we are interested in | | |
| 243 | andl %uart000000,d1 | see if VRTX interrupt | | |
| 244 | jeq rtscops_serial_isr | no? go to RTscops | | |
| 245 | movl a0,d1 | load into reg | | |
| 246 | btst %CHAR_RX_A,d1 | check for CHAR_RX_A | | |
| 247 | bneq not_rx_a | nothing rxd on channel a | | |
| 248 | movl %STATUS_REC_A,a0 | get the status | | |
| 249 | movl a0,d0 | status is in 24-31 | | |
| 250 | movl %DUART_RX_TX_REC_A,a0 | get the byte | | |
| 251 | movl a0,d1 | byte is in 24-31 | | |
| 252 | andl %uart000000,d0 | see if error to rxd, break or framing | | |
| 253 | bneq err_rx_a | process error | | |
| 254 | movsq %24,d0 | shift 24 bits | | |
| 255 | asrl d0,d1 | d1 bits 0-7 = data | | |
| 256 | | | | |
| 257 | cmpl %XOFF,d1 | Flow control check for XON & XOFF | | |
| 258 | jne not_xoff | | | |
| 259 | movb 1,XOFF | | | |
| 260 | jar txa_exit | | | |
| 261 | bneq not_tx_a | | | |
| 262 | not_xoff: | | | |
| 263 | cmpl %XON,d1 | | | |
| 264 | jne not_xon | | | |
| 265 | movb 0,XOFF | | | |
| 266 | bneq not_tx_a | | | |
| 267 | not_xon: | | | |
| 268 | | | | |
| 269 | movl %UI_RXCH,d0 | | | |
| 270 | trap %VRTX | | | |
| 271 | not_rx_a: | | | |
| 272 | movl %DUART_SR,a0 | get status | | |
| 273 | movl a0,d1 | load into reg | | |
| 274 | btst %CHAR_TX_A,d1 | check for CHAR_TX_A | | |
| 275 | bneq not_tx_a | nothing to tx on channel a | | |
| 276 | jar txa_isr | go and transmit | | |
| 277 | not_tx_a: | | | |
| 278 | movl %pc,a0 | restore address register | | |
| 279 | movl %pc,d1 | | | |
| 280 | movl %UI_EXIT,d0 | | | |
| 281 | trap %VRTX | | | |
| 282 | err_rx_a: | | | |
| 283 | andl %uart000000,d0 | see if break | | |
| 284 | bneq not_break | not break | | |
| 285 | movsq %24,d0 | shift 24 bits | | |
| 286 | asrl d0,d1 | d1 bits 0-7 = data | | |
| 287 | bneq not_tx_a | not break start | | |
| 288 | jar txa_rx_brk | process break | | |
| 289 | bneq not_tx_a | continue | | |
| 290 | | | | |
| 291 | END txa_isr | | | |
| 292 | | | | |
| 293 | | | | |
| 294 | Channel A Serial Port - TX buffer empty | | | |
| 295 | | | | |
| 296 | global txa_isr, Vrtx_txa | | | |
| 297 | txa_isr: | | | |
| 298 | cmpl %x1,XOFF | Check flow control is not on | | |
| 299 | jne no_flow_ctrl | | | |
| 300 | bneq txa_exit | | | |
| 301 | no_flow_ctrl: | | | |
| 302 | | | | |
| 303 | movl %UI_TXCH,d0 | | | |
| 304 | trap %VRTX | | | |
| 305 | cmpl %0,d0 | | | |
| 306 | bneq txa_exit | | | |
| 307 | | | | |
| 308 | | | | |
| 309 | ----- This is the VRTX TX routine ----- | | | |
| 310 | ----- d1 = data to TX ----- | | | |
| 311 | ----- save A0 & D0 ----- | | | |
| 312 | ----- This is the hook for VRTX to resume or start transmitting chars ----- | | | |
| 313 | Vrtx_txa: | | | |
| 314 | movl d0,%pc | get address of register | | |
| 315 | movl a0,%pc | shift 24 bits to get byte in upper bits | | |
| 316 | movl %DUART_RX_TX_REC_A,a0 | d1 bits 00-07 = data | | |
| 317 | movsq %24,d0 | byte is in 24-31 after shift | | |
| 318 | asrl d0,d1 | get status reg contents | | |
| 319 | movl d1,a0 | this is a write only reg. so read from memory | | |
| 320 | movl %DUART_SR,a0 | d1 = 03 enable tx ints | | |
| 321 | movl %uart_mask,d1 | save in memory | | |
| 322 | orl %uart000000,d1 | load into reg | | |
| 323 | movl d1,%uart_mask | restore registers | | |
| 324 | movl d1,a0 | | | |
| 325 | movl %pc,a0 | | | |
| 326 | movl %pc,d0 | | | |
| 327 | movl %pc,d0 | | | |
| 328 | txa_exit: | | | |
| 329 | movl %DUART_SR,a0 | get status reg contents | | |
| 330 | movl %uart_mask,d1 | this is a write only reg. so read from memory | | |
| 331 | andl %uart000000,d1 | d1 = disable tx ints | | |
| 332 | movl d1,%uart_mask | save in memory | | |
| 333 | movl d1,a0 | load into reg | | |
| 334 | movl d1,a0 | | | |
| 335 | movl d1,a0 | | | |
| 336 | END txa_isr | | | |
| 337 | | | | |
| 338 | RTscops serial int. on chan B | | | |
| 339 | | | | |
| 340 | rtscops_serial_isr: | | | |
| 341 | movl a0,d1 | load into reg | | |
| 342 | btst %CHAR_RX_B,d1 | check for CHAR_RX_B | | |
| 343 | bneq not_rx_b | nothing rxd on channel b | | |
| 344 | jar rxb_isr | receive character | | |
| 345 | not_rx_b: | | | |
| 346 | movl %DUART_SR,a0 | get status | | |
| 347 | movl a0,d1 | load into reg | | |
| 348 | btst %CHAR_TX_B,d1 | check for CHAR_TX_B | | |
| 349 | bneq not_tx_b | nothing can be transmitted on channel b | | |
| 350 | jar rxb_isr | transmit character | | |
| 351 | not_tx_b: | | | |
| 352 | movl %pc,a0 | restore address register | | |
| 353 | movl %pc,d1 | | | |
| 354 | movl %UI_EXIT,d0 | | | |
| 355 | trap %VRTX | | | |
| 356 | END rxb_isr | | | |
| 357 | | | | |
| 358 | | | | |
| 359 | ----- This is the RTSCOPS TX routine ----- | | | |
| 360 | ----- d1 = data to TX ----- | | | |

| Copyright 1989 Logic Modeling Systems | | FILE bsp.diags/bsp.assem.s | DATE 5/23/89 TIME 4:41:07 pm | PAGE # 4/4 |
|--|--|-------------------------------|---|---------------|
| LINE # | TEXT | | | |
| 361 | ----- save A0 & D0 ----- | | | |
| 362 | ----- This is the hook for RTSCOPE to resume or start transmitting chars ----- | | | |
| 363 | | | | |
| 364 | .globl _Vrtx_tx | | | |
| 365 | _Vrtx_tx: | | | |
| 366 | rts | | | |
| 367 | ----- txs txs_isr ----- | | | |
| 368 | | | | |
| 369 | .globl _rtscope_in_poll, _rtscope_out_poll | | | |
| 370 | _rtscope_in_poll: | | | |
| 371 | movl | \$0, %d0 | | |
| 372 | movl | \$1, %d1 | | |
| 373 | movl | 0xfffffff, %d0 | status | |
| 374 | cmpl | 0, %d0 | | |
| 375 | btl | inc_not_tx_b_poll | | |
| 376 | cmpl | 0, %d0 | | |
| 377 | bne | not_time_for_C | | |
| 378 | movl | 0x57, %d0 | | |
| 379 | bne | inc_not_tx_b_poll | | |
| 380 | not_time_for_C: | | | |
| 381 | cmpl | 0, %d0 | | |
| 382 | bne | not_time_for_O | | |
| 383 | movl | 0x57, %d0 | | |
| 384 | bne | inc_not_tx_b_poll | | |
| 385 | not_time_for_O: | | | |
| 386 | cmpl | 0, %d0 | | |
| 387 | bne | not_time_for_0xD | | |
| 388 | movl | 0x57, %d0 | | |
| 389 | bne | inc_not_tx_b_poll | | |
| 390 | not_time_for_0xD: | | | |
| 391 | movl | 0, %d0 | get status reg contents | |
| 392 | movl | 0, %d1 | load into reg | |
| 393 | btl | inc_not_tx_b_poll | check for CHAR_TX_B | |
| 394 | bneq | not_tx_b_poll | nothing rcvd on channel b | |
| 395 | movl | 0, %d0 | get the byte | |
| 396 | movl | 0x57, %d0 | byte is in 24-31 | |
| 397 | movl | 0x57, %d0 | shift 24 bits | |
| 398 | sarl | 0x10, %d0 | d1 bits 0-7 = data | |
| 399 | | | | |
| 400 | cmpl | 0, %d0 | check for " | |
| 401 | jne | not_poll_cache_disable | cache disable | |
| 402 | movl | 0, %d0 | | |
| 403 | orl | 0, %d0 | | |
| 404 | movl | 0, %d0 | Clear and disable instruction cache | |
| 405 | jmp | not_tx_b_poll | | |
| 406 | not_poll_cache_disable: | | | |
| 407 | | | | |
| 408 | not_tx_b_poll: | | | |
| 409 | movl | 0, %d0 | restore registers | |
| 410 | movl | 0, %d1 | | |
| 411 | rts | | | |
| 412 | inc_not_tx_b_poll: | | | |
| 413 | addl | 0, %d0 | restore registers | |
| 414 | movl | 0, %d0 | | |
| 415 | movl | 0, %d1 | | |
| 416 | rts | | | |
| 417 | | | | |
| 418 | | | | |
| 419 | | | | |
| 420 | _rtscope_out_poll: | | | |
| 421 | movl | 0, %d0 | | |
| 422 | movl | 0, %d1 | | |
| 423 | movl | 0, %d0 | get status | |
| 424 | movl | 0, %d1 | load into reg | |
| 425 | movl | 0, %d0 | check for CHAR_TX_B | |
| 426 | btl | inc_not_tx_b_poll | nothing can be transmitted on channel b | |
| 427 | bneq | not_tx_b_poll | get address of register | |
| 428 | movl | 0, %d0 | shift 24 bits to get byte in upper bits | |
| 429 | movl | 0, %d1 | d0 bits 00-07 = data | |
| 430 | sarl | 0x10, %d0 | byte is in 24-31 after shift | |
| 431 | movl | 0, %d0 | get status reg contents | |
| 432 | movl | 0, %d1 | this is a write only reg. so read from memory | |
| 433 | movl | 0, %d0 | d1 = 01 enable tx ints | |
| 434 | orl | 0, %d0 | save in memory | |
| 435 | movl | 0, %d0 | load into reg | |
| 436 | movl | 0, %d0 | successfully tx | |
| 437 | movl | 0, %d0 | restore registers | |
| 438 | movl | 0, %d0 | | |
| 439 | movl | 0, %d1 | | |
| 440 | rts | | | |
| 441 | | | | |
| 442 | not_tx_b_poll: | | | |
| 443 | movl | 0, %d0 | no tx took place | |
| 444 | movl | 0, %d0 | restore registers | |
| 445 | movl | 0, %d1 | | |
| 446 | rts | | | |
| 447 | | | | |
| 448 | ----- This is the BUS ERROR TX routine ----- | | | |
| 449 | ----- d1 = data to TX ----- | | | |
| 450 | ----- save A0 & D0 ----- | | | |
| 451 | | | | |
| 452 | .globl _bus_error_tx | | | |
| 453 | _bus_error_tx: | | | |
| 454 | link | 0, %d0 | | |
| 455 | movl | 0, %d0 | | |
| 456 | movl | 0, %d0 | | |
| 457 | not_ready_tx_a: | | | |
| 458 | movl | 0, %d0 | get status | |
| 459 | movl | 0, %d1 | load into reg | |
| 460 | btl | inc_not_tx_b_poll | check for CHAR_TX_A | |
| 461 | bneq | not_ready_tx_a | nothing to tx on channel a | |
| 462 | movl | 0, %d0 | data to tx | |
| 463 | movl | 0, %d1 | get address of register | |
| 464 | movl | 0, %d0 | shift 24 bits to get byte in upper bits | |
| 465 | sarl | 0x10, %d0 | d1 bits 00-07 = data | |
| 466 | movl | 0, %d0 | byte is in 24-31 after shift | |
| 467 | movl | 0, %d0 | restore registers | |
| 468 | movl | 0, %d0 | | |
| 469 | movl | 0, %d1 | | |
| 470 | unlk | 0 | | |
| 471 | rts | | | |
| 472 | ----- Counter/Timer Interrupt ----- | | | |
| 473 | | | | |
| 474 | .globl _ct_isr | | | |
| 475 | _ct_isr: | | | |
| 476 | | | | |
| 477 | | | | |
| 478 | This interrupt occurs every 5 ms | | | |
| 479 | we are using mode 0. | | | |
| 480 | | | | |

| Copyright 1989 Logic Modeling Systems | | FILE bsp.diags/bsp.assem.s | DATE 5/23/89 | PAGE # 5/5 |
|--|---|---|-----------------|---------------|
| LINE # | | TEXT | | |
| 481 | movl d0,ap0- | | | |
| 482 | movl d0,ap0- | | | |
| 483 | movl d1,ap0- | | | |
| 484 | movl d1,ap0- | | | |
| 485 | | | | |
| 486 | | | | |
| 487 | | | | |
| 488 | -- disable interrupts | | | |
| 489 | | | | |
| 490 | movl PCPS_INTX_REG,a0 | | | |
| 491 | movb a0,d1 | | | |
| 492 | andb \$0x7f,d1 | | | |
| 493 | | | | |
| 494 | | | | |
| 495 | -- reload tick value | | | |
| 496 | | | | |
| 497 | movl PTIMER_CONTROL,a0 | | | |
| 498 | movl SREAR_COUNTER1,a0 | | | |
| 499 | movl PTIMER2_COUNT,a0 | | | |
| 500 | movl a0,d1 | | | |
| 501 | movl \$24,d0 | | | |
| 502 | sarl d0,d1 | d1 bits 0-7 = low order data | | |
| 503 | andl \$0xf,d1 | | | |
| 504 | movl a0,d0 | | | |
| 505 | sarl d0,d0 | | | |
| 506 | sarl d0,d0 | d0 bits 8-f = high order data | | |
| 507 | andl \$0xf,d0 | | | |
| 508 | addl d1,d0 | | | |
| 509 | andl \$0xffff,d0 | mask off count | | |
| 510 | addw PTIMER2_TOTAL,d0 | | | |
| 511 | call \$0,d0 | | | |
| 512 | call \$0,d0 | | | |
| 513 | movl d0,d1 | | | |
| 514 | call \$0,d0 | | | |
| 515 | movl d0,a0 | | | |
| 516 | movl d1,a0 | | | |
| 517 | movl PTIMER2_LOW_COUNT,a0 | | | |
| 518 | movl PTIMER2_HIGH_COUNT,a0 | of ticks since beginning of time | | |
| 519 | | | | |
| 520 | -- Enable interrupts | | | |
| 521 | | | | |
| 522 | movl PCPS_INTX_REG,a0 | | | |
| 523 | btsb \$7,d1 | | | |
| 524 | bqz disabled_ints | | | |
| 525 | crb \$0x00,a0 | | | |
| 526 | disabled_ints: | | | |
| 527 | addl \$1,_in_tick | This statement saves us a VRTX system call to determine no. | | |
| 528 | addw \$1,housekeeping | is it 1 second yet | | |
| 529 | cmpr \$200,housekeeping | no. of ticks before we run housekeeping | | |
| 530 | jbe no_housekeeping | | | |
| 531 | clr housekeeping | reset counter | | |
| 532 | movl \$SC_IPOST,d0 | post to semaphore | | |
| 533 | movl timer_semaphore,d1 | semaphore ID number | | |
| 534 | trap \$VRTX | | | |
| 535 | no_housekeeping: | | | |
| 536 | | | | |
| 537 | ----- Make VI_TIMER Call to Vrtx | | | |
| 538 | | | | |
| 539 | | | | |
| 540 | - inform VRTX of tick | | | |
| 541 | | | | |
| 542 | movl \$VI_TIMER,d0 | | | |
| 543 | trap \$VRTX | | | |
| 544 | | | | |
| 545 | movl ap0+,d1 | restore registers | | |
| 546 | movl ap0+,d1 | restore registers | | |
| 547 | movl ap0+,a0 | restore registers | | |
| 548 | | | | |
| 549 | | | | |
| 550 | | | | |
| 551 | ----- Perform VI_EXIT from ISR through Vrtx | | | |
| 552 | | | | |
| 553 | movl \$VI_EXIT,d0 | | | |
| 554 | trap \$VRTX | | | |
| 555 | | | | |
| 556 | END of_isr | | | |
| 557 | | | | |
| 558 | | | | |
| 559 | ----- PROBE BUS ERROR HANDLER | | | |
| 560 | | | | |
| 561 | .globl _bus_isr | | | |
| 562 | _bus_isr: | | | |
| 563 | movl sp,_bus_error_address | | | |
| 564 | | | | |
| 565 | moveml \$0xffff,ap0- | dont save a7 save d1-d7,a0-a6 | | |
| 566 | jmr _bus_error | | | |
| 567 | moveml ap0+,\$0x7fff | restore above regs. | | |
| 568 | rts | | | |
| 569 | | | | |
| 570 | ----- DIAG BUS ERROR HANDLER | | | |
| 571 | | | | |
| 572 | .globl _diag_bus_isr | | | |
| 573 | _diag_bus_isr: | | | |
| 574 | movl sp,_bus_error_address | | | |
| 575 | | | | |
| 576 | moveml \$0xffff,ap0- | dont save a7 save d1-d7,a0-a6 | | |
| 577 | jmr _diag_bus_error | | | |
| 578 | moveml ap0+,\$0x7fff | restore above regs. | | |
| 579 | rts | | | |
| 580 | | | | |
| 581 | | | | |
| 582 | ----- TRAP ROUTINES for RTscope and VRTX | | | |
| 583 | | | | |
| 584 | | | | |
| 585 | | | | |
| 586 | .globl _trp_vrtx | | | |
| 587 | _trp_vrtx: | | | |
| 588 | | | | |
| 589 | trap \$VRTX | | | |
| 590 | rts | | | |
| 591 | | | | |
| 592 | .globl _trp_rtscope | | | |
| 593 | _trp_rtscope: | | | |
| 594 | | | | |
| 595 | trap \$RTSCOPE | | | |
| 596 | rts | | | |
| 597 | | | | |
| 598 | | | | |
| 599 | | | | |
| 600 | | | | |

Copyright 1989
Logic Modeling Systems

FILE
bsp.diags/bsp.assem.s

DATE 5/23/89
TIME 4:41:07 pm

PAGE #
6/6

```

LINE #      TEXT
601      ----- RESET_BSD
602      cause a board reset.
603
604      ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
605
606      .globl _reset_brd
607      _reset_brd:
608      reset
609      jar    bsp_start
610
611      board reset occurs here
612
613      rts
614
615
616      .globl _do_nothing
617      _do_nothing:
618      rts
619
620
621      ----- Create a supervisory task -----
622      err = sc_tcreate_supv_set_boot( task_address, SUPV\USER task , task_id, task priority)
623      if( err != 0 ) FAILURE;
624
625      .globl _sc_tcreate_supv_set_boot
626      _sc_tcreate_supv_set_boot:
627      link    a5, $0
628      movl    a0, spc
629      movl    d1, spc
630      movl    d1, spc
631      movl    d1, spc
632      movl    $0, d1
633      movl    $0, d1
634      movl    a6($1), a0      task address
635      movl    a6(0x0c), d3    task mode
636      movl    a6(0x10), d2    task id
637      movl    a6(0x14), d1    task priority
638      movl    $SC_TCREATE, d0  System call code
639      trap    $VRTX          Call VRTX
640      movl    spc+, d3
641      movl    spc+, d2
642      movl    spc+, d1
643      movl    spc+, a0
644      halt    a0
645      rts
646
647
648      .globl _disable_cache
649      _disable_cache:
650      movl    $8, d0
651      movc    d0, cacr
652      rts

```

Clear and disable instruction cache

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

bsp.diags/bsp.c

DATE

5/23/89

PAGE #

TIME

4:41:07 pm

1/7

```

1  // SCCS ID: bsp.c rev 3.1, 4/24/89 at 07:55:02
2  /*
3  ** This is part of the board support package
4  **
5  ** Make sure that both bsp.lm500/bsp.c and bsp.diags/bsp.c
6  ** are exactly the same.
7  */
8
9  #include "common.h"
10 #include "task.h"
11 #include "uart.h"
12 #include "lm_rst_wr.h"
13 #include "lm500.h"
14 #include "cpu.h"
15 #include "cpu_reg.h"
16 #include "mem_err.h"
17 #include "swram.h"
18 #include "id.h"
19
20 /* External routine definitions */
21 extern int init_sys();
22 extern void serial_isr();
23 extern void parity_isr();
24 extern void ct_isr();
25 extern void reset_cpu();
26 extern void other_isr();
27 extern void error_isr();
28 extern void vtx_task();
29 extern void rtscop_in_poll();
30 extern void rtscop_out_poll();
31 extern void void_id_load();
32 extern void tpx_vtx();
33 extern void tpx_rtscop();
34 extern void u_short_in_swram_access();
35 extern void u_long_network_timeout();
36 extern void reset_hw();
37
38 #define MICROSEC PER_SECOND 1000
39
40 /*----- The Globals -----*/
41
42 /* Init_sys */
43 /* Activate RTSCOP and VTX */
44 /* STOPS */
45 /* 1. Setup interrupt vector table */
46 /* 2. Create VTX conf table */
47 /* 3. Create RTSCOP conf table */
48 /* 4. Create component table */
49 /* 5. Initialize VTX */
50 /* 6. Initialize RTSCOP */
51 /* 07. Setup 8254 timers */
52 /* 08. Initialize UART chan A */
53 /* 09. Enable CPU reg. */
54 /* 10. Partition, commit & malloc */
55 /* 11. RTSCOP go command */
56
57 unsigned long rtscop = 0;
58 u_long total_malloc_size = 0;
59 u_long available_malloc_size;
60 ID_PROG_CPU id_prog;
61
62 /* Initialize routine */
63
64 init_sys()
65 {
66     long err = 0;
67     char baud;
68     u_short short_network_timeout;
69     cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG;
70
71     /*
72     ** set modeler state to booted
73     */
74     modeler_state = BOOTED;
75
76     /* Turn off load led */
77
78     p_cpu_ctl_reg->net_load_led = not_LED_OFF;
79
80     /* shutdown and reset lanes. */
81     p_cpu_ctl_reg->net_lane_reset = 1;
82
83     shutdown_and_reset_lanes();
84
85     /* calculate total partition size */
86
87     total_malloc_size = (u_long)CPU_RAM_SIZE - (u_long)(load);
88     available_malloc_size = total_malloc_size;
89
90     setup_ivt(); /* Setup the interrupt table */
91     init_rtscop(); /* Create VTX conf. table */
92     init_rtscop_conf(); /* Create RTSCOP conf. table */
93     init_vtx(); /* Create component table */
94     err = vtx_init(); /* Init VTX */
95     if( err )
96     {
97         /* error, wait in loop */
98         for( ;; )
99             reset_cpu(SUICIDE);
100     }
101
102 #ifdef RTSCOP_DIAGS_DEBUG
103     err = rtscop_init( &(rtscop->r_conf) );
104     if( err )
105     {
106         /* error, wait in loop */
107         for( ;; )
108             reset_cpu(SUICIDE);
109     }
110 #endif
111
112 /*
113 ** Initialize board devices
114 */
115 init_all_timer(); /* initialize all 3 timers */
116
117 /*
118 ** validate NVRam, setup baud rate, it should be valid
119 */
120 (void)lm_swram_access((char *)0, (u_long)0, (u_long)0, MEMORY_VALIDATE, (u_long *) &err);
121
122 /* set Modeler state to booted

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
bsp.diags/bsp.c

DATE 5/23/89
TIME 4:41:07 pm

PAGE #
2/8

```

LINE #          SOURCE TEXT
121          /*
122          (void)lm_vram_access(lmodeler_state, MODELER_STATE, SIZEOF_MODELER_STATE, MEMORY_WRITE, (u_long *) &err);
123          Uarta_inx();
124          Uartb_inx();
125          /*
126          set network timeout
127          /*
128          if(lm_vram_access(&short_network_timeout, NETWORK_TIMEOUT, SIZEOF_NETWORK_TIMEOUT, MEMORY_READ, (u_long *) &err) == FAILURE)
129          {
130              sys_out("Unable to read network timeout\n");
131          }
132          /*
133          Turn seconds to milliseconds
134          /*
135          network_timeout = short_network_timeout;
136          network_timeout *= NSECONDS_PER_SECOND;
137          /*
138          read id from
139          id_load((u_char *)CPU_ID_PROM, (u_char *)(&id_prom)); /* Fetch id prom */
140          enable_cpu_regs(); /* enable CPU regs */
141          rts_go(); /* Execute RTscope GO command */
142          /*
143          /*
144          /* SHOULD NOT COME BACK. If it does ??????????
145          /* do nothing.
146          /*
147          for(;;)
148          {
149              reset_cpu(SUICIDE);
150          }
151          /* end init_sys */
152          /*
153          /*
154          /*
155          /*
156          /*
157          /*
158          /*
159          /*
160          /*
161          /*
162          /*
163          /*
164          /*
165          /*
166          /*
167          /*
168          /*
169          /*
170          /*
171          /*
172          /*
173          /*
174          /*
175          /*
176          /*
177          /*
178          /*
179          /*
180          /*
181          /*
182          /*
183          /*
184          /*
185          /*
186          /*
187          /*
188          /*
189          /*
190          /*
191          /*
192          /*
193          /*
194          /*
195          /*
196          /*
197          /*
198          /*
199          /*
200          /*
201          /*
202          /*
203          /*
204          /*
205          /*
206          /*
207          /*
208          /*
209          /*
210          /*
211          /*
212          /*
213          /*
214          /*
215          /*
216          /*
217          /*
218          /*
219          /*
220          /*
221          /*
222          /*
223          /*
224          /*
225          /*
226          /*
227          /*
228          /*
229          /*
230          /*
231          /*
232          /*
233          /*
234          /*
235          /*
236          /*
237          /*
238          /*
239          /*
240          /*
241          /*
242          /*
243          /*
244          /*
245          /*
246          /*
247          /*
248          /*
249          /*
250          /*
251          /*
252          /*
253          /*
254          /*
255          /*
256          /*
257          /*
258          /*
259          /*
260          /*
261          /*
262          /*
263          /*
264          /*
265          /*
266          /*
267          /*
268          /*
269          /*
270          /*
271          /*
272          /*
273          /*
274          /*
275          /*
276          /*
277          /*
278          /*
279          /*
280          /*
281          /*
282          /*
283          /*
284          /*
285          /*
286          /*
287          /*
288          /*
289          /*
290          /*
291          /*
292          /*
293          /*
294          /*
295          /*
296          /*
297          /*
298          /*
299          /*
300          /*
301          /*
302          /*
303          /*
304          /*
305          /*
306          /*
307          /*
308          /*
309          /*
310          /*
311          /*
312          /*
313          /*
314          /*
315          /*
316          /*
317          /*
318          /*
319          /*
320          /*
321          /*
322          /*
323          /*
324          /*
325          /*
326          /*
327          /*
328          /*
329          /*
330          /*
331          /*
332          /*
333          /*
334          /*
335          /*
336          /*
337          /*
338          /*
339          /*
340          /*
341          /*
342          /*
343          /*
344          /*
345          /*
346          /*
347          /*
348          /*
349          /*
350          /*
351          /*
352          /*
353          /*
354          /*
355          /*
356          /*
357          /*
358          /*
359          /*
360          /*
361          /*
362          /*
363          /*
364          /*
365          /*
366          /*
367          /*
368          /*
369          /*
370          /*
371          /*
372          /*
373          /*
374          /*
375          /*
376          /*
377          /*
378          /*
379          /*
380          /*
381          /*
382          /*
383          /*
384          /*
385          /*
386          /*
387          /*
388          /*
389          /*
390          /*
391          /*
392          /*
393          /*
394          /*
395          /*
396          /*
397          /*
398          /*
399          /*
400          /*
401          /*
402          /*
403          /*
404          /*
405          /*
406          /*
407          /*
408          /*
409          /*
410          /*
411          /*
412          /*
413          /*
414          /*
415          /*
416          /*
417          /*
418          /*
419          /*
420          /*
421          /*
422          /*
423          /*
424          /*
425          /*
426          /*
427          /*
428          /*
429          /*
430          /*
431          /*
432          /*
433          /*
434          /*
435          /*
436          /*
437          /*
438          /*
439          /*
440          /*
441          /*
442          /*
443          /*
444          /*
445          /*
446          /*
447          /*
448          /*
449          /*
450          /*
451          /*
452          /*
453          /*
454          /*
455          /*
456          /*
457          /*
458          /*
459          /*
460          /*
461          /*
462          /*
463          /*
464          /*
465          /*
466          /*
467          /*
468          /*
469          /*
470          /*
471          /*
472          /*
473          /*
474          /*
475          /*
476          /*
477          /*
478          /*
479          /*
480          /*
481          /*
482          /*
483          /*
484          /*
485          /*
486          /*
487          /*
488          /*
489          /*
490          /*
491          /*
492          /*
493          /*
494          /*
495          /*
496          /*
497          /*
498          /*
499          /*
500          /*
501          /*
502          /*
503          /*
504          /*
505          /*
506          /*
507          /*
508          /*
509          /*
510          /*
511          /*
512          /*
513          /*
514          /*
515          /*
516          /*
517          /*
518          /*
519          /*
520          /*
521          /*
522          /*
523          /*
524          /*
525          /*
526          /*
527          /*
528          /*
529          /*
530          /*
531          /*
532          /*
533          /*
534          /*
535          /*
536          /*
537          /*
538          /*
539          /*
540          /*
541          /*
542          /*
543          /*
544          /*
545          /*
546          /*
547          /*
548          /*
549          /*
550          /*
551          /*
552          /*
553          /*
554          /*
555          /*
556          /*
557          /*
558          /*
559          /*
560          /*
561          /*
562          /*
563          /*
564          /*
565          /*
566          /*
567          /*
568          /*
569          /*
570          /*
571          /*
572          /*
573          /*
574          /*
575          /*
576          /*
577          /*
578          /*
579          /*
580          /*
581          /*
582          /*
583          /*
584          /*
585          /*
586          /*
587          /*
588          /*
589          /*
590          /*
591          /*
592          /*
593          /*
594          /*
595          /*
596          /*
597          /*
598          /*
599          /*
600          /*
601          /*
602          /*
603          /*
604          /*
605          /*
606          /*
607          /*
608          /*
609          /*
610          /*
611          /*
612          /*
613          /*
614          /*
615          /*
616          /*
617          /*
618          /*
619          /*
620          /*
621          /*
622          /*
623          /*
624          /*
625          /*
626          /*
627          /*
628          /*
629          /*
630          /*
631          /*
632          /*
633          /*
634          /*
635          /*
636          /*
637          /*
638          /*
639          /*
640          /*
641          /*
642          /*
643          /*
644          /*
645          /*
646          /*
647          /*
648          /*
649          /*
650          /*
651          /*
652          /*
653          /*
654          /*
655          /*
656          /*
657          /*
658          /*
659          /*
660          /*
661          /*
662          /*
663          /*
664          /*
665          /*
666          /*
667          /*
668          /*
669          /*
670          /*
671          /*
672          /*
673          /*
674          /*
675          /*
676          /*
677          /*
678          /*
679          /*
680          /*
681          /*
682          /*
683          /*
684          /*
685          /*
686          /*
687          /*
688          /*
689          /*
690          /*
691          /*
692          /*
693          /*
694          /*
695          /*
696          /*
697          /*
698          /*
699          /*
700          /*
701          /*
702          /*
703          /*
704          /*
705          /*
706          /*
707          /*
708          /*
709          /*
710          /*
711          /*
712          /*
713          /*
714          /*
715          /*
716          /*
717          /*
718          /*
719          /*
720          /*
721          /*
722          /*
723          /*
724          /*
725          /*
726          /*
727          /*
728          /*
729          /*
730          /*
731          /*
732          /*
733          /*
734          /*
735          /*
736          /*
737          /*
738          /*
739          /*
740          /*
741          /*
742          /*
743          /*
744          /*
745          /*
746          /*
747          /*
748          /*
749          /*
750          /*
751          /*
752          /*
753          /*
754          /*
755          /*
756          /*
757          /*
758          /*
759          /*
760          /*
761          /*
762          /*
763          /*
764          /*
765          /*
766          /*
767          /*
768          /*
769          /*
770          /*
771          /*
772          /*
773          /*
774          /*
775          /*
776          /*
777          /*
778          /*
779          /*
780          /*
781          /*
782          /*
783          /*
784          /*
785          /*
786          /*
787          /*
788          /*
789          /*
790          /*
791          /*
792          /*
793          /*
794          /*
795          /*
796          /*
797          /*
798          /*
799          /*
800          /*
801          /*
802          /*
803          /*
804          /*
805          /*
806          /*
807          /*
808          /*
809          /*
810          /*
811          /*
812          /*
813          /*
814          /*
815          /*
816          /*
817          /*
818          /*
819          /*
820          /*
821          /*
822          /*
823          /*
824          /*
825          /*
826          /*
827          /*
828          /*
829          /*
830          /*
831          /*
832          /*
833          /*
834          /*
835          /*
836          /*
837          /*
838          /*
839          /*
840          /*
841          /*
842          /*
843          /*
844          /*
845          /*
846          /*
847          /*
848          /*
849          /*
850          /*
851          /*
852          /*
853          /*
854          /*
855          /*
856          /*
857          /*
858          /*
859          /*
860          /*
861          /*
862          /*
863          /*
864          /*
865          /*
866          /*
867          /*
868          /*
869          /*
870          /*
871          /*
872          /*
873          /*
874          /*
875          /*
876          /*
877          /*
878          /*
879          /*
880          /*
881          /*
882          /*
883          /*
884          /*
885          /*
886          /*
887          /*
888          /*
889          /*
890          /*
891          /*
892          /*
893          /*
894          /*
895          /*
896          /*
897          /*
898          /*
899          /*
900          /*
901          /*
902          /*
903          /*
904          /*
905          /*
906          /*
907          /*
908          /*
909          /*
910          /*
911          /*
912          /*
913          /*
914          /*
915          /*
916          /*
917          /*
918          /*
919          /*
920          /*
921          /*
922          /*
923          /*
924          /*
925          /*
926          /*
927          /*
928          /*
929          /*
930          /*
931          /*
932          /*
933          /*
934          /*
935          /*
936          /*
937          /*
938          /*
939          /*
940          /*
941          /*
942          /*
943          /*
944          /*
945          /*
946          /*
947          /*
948          /*
949          /*
950          /*
951          /*
952          /*
953          /*
954          /*
955          /*
956          /*
957          /*
958          /*
959          /*
960          /*
961          /*
962          /*
963          /*
964          /*
965          /*
966          /*
967          /*
968          /*
969          /*
970          /*
971          /*
972          /*
973          /*
974          /*
975          /*
976          /*
977          /*
978          /*
979          /*
980          /*
981          /*
982          /*
983          /*
984          /*
985          /*
986          /*
987          /*
988          /*
989          /*
990          /*
991          /*
992          /*
993          /*
994          /*
995          /*
996          /*
997          /*
998          /*
999          /*
1000         /*

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

bsp.diags/bsp.c

DATE

5/23/89

PAGE #

TIME

4:41:07 pm

3/9

```

LINE #          SOURCE TEXT
241
242
243 /*-----*/
244 /*          init_cvt          */
245 /* Create component configuration table */
246 /*-----*/
247
248 init_cvt()
249 {
250     Cvt_TBL      *ct = (Cvt_TBL *)(&cfbtle->cvt);
251     short
252
253     ct->hr_max = HR_MAX,          /* Ready System highest component no. */
254     ct->usr_max = USR_MAX,        /* User highest component no. */
255
256     for( i = 0, i < 6; i++ )      /* Reserved */
257         ct->real[ i ] = 0,
258     for( i = 0, i < 8; i++ )      /* Reserved */
259         ct->real2[ i ] = 0,
260
261     ct->compl_code = 0,           /* No IOX */
262     ct->compl_work = 0,           /* No IOX */
263     ct->compl_code = 0,           /* No FPG */
264     ct->compl_work = 0,           /* No FPG */
265
266 } /* end init_cvt */
267
268
269
270
271
272 /*-----*/
273 /*          setup_ivt          */
274 /* Setup interrupt vector table */
275 /*-----*/
276
277 setup_ivt()
278 {
279     unsigned long *ivt = (unsigned long *)VEC_BASE_ADDRESS,
280
281     /*
282     ** The bus handler
283     */
284     *(ivt + VEC_BUS_ERROR) = (unsigned long) bus_isr,
285     *(ivt + VEC_ADDRESS_ERROR) = (unsigned long) bus_isr,
286     *(ivt + VEC_ILLEGAL_INSTRUCTION) = (unsigned long) bus_isr,
287
288     /*
289     ** Set up rtoscope entry
290     */
291     *(ivt + 0x11) = (RTS_BASE + RTS_ENTRY), /* RTscope entry point */
292
293     /*
294     ** Set up Vtrix vectors
295     */
296     *(ivt + 0x40) = (long)&(cfbtle->v_cft), /* Config Table */
297     *(ivt + 0x20) = VTRI_BASE,             /* VTRI Base */
298
299     /*
300     ** Set up Serial port interrupt vectors
301     */
302     /* Tx/Rx ISR handler address to IVT auto vector level 3 */
303     *(ivt + VEC_DUART_INTERRUPT) = (long)serial_isr,
304
305     /*
306     ** Set up Counter/Timer interrupt vectors
307     */
308     /* Count/Timer ISR address to IVT 25, Auto vector level 4 */
309     *(ivt + VEC_TIMER_INTERRUPT) = (unsigned long)ct_isr,
310
311     /*
312     ** Set up ethernet interrupt vector
313     */
314     /* ethernet ISR address to IVT 10, Auto vector level 6 */
315     *(ivt + VEC_LANCE_DOORBELL_INTE) = (unsigned long)ether_isr,
316
317     /*
318     ** Clock board error ISR
319     */
320     *(ivt + VEC_Cb_INTERRUPT) = (unsigned long)error_isr,
321
322     /*
323     ** Parity ISR, IVT 31, MMIO
324     */
325     *(ivt + VEC_PARITY_INTERRUPT) = (unsigned long)parity_isr,
326
327 } /* end setup_ivt */
328
329
330
331
332 /*
333 ** Enable cpu control registers
334 */
335 enable_cpu_reg()
336 {
337     cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG,
338
339     /*
340     ** enable timer gates, enable the timer interrupt
341     */
342     p_cpu_ctl_reg->timer_gate_0 = 1,
343     p_cpu_ctl_reg->timer_gate_1 = 1,
344     p_cpu_ctl_reg->timer_intr_ena_2 = 1,
345
346     /*
347     ** enable parity intr
348     */
349     p_cpu_ctl_reg->parity_intr_ena = 1,
350
351     /*
352     ** enable interrupts
353     */
354     p_cpu_ctl_reg->global_intr_ena = 1,
355
356 }
357
358 /*
359 ** reset the CPU
360 */
361 void
362 reset_cpu(reset_modeler_state)
363 char reset_modeler_state,

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

bsp.diags/bsp.c

DATE

5/23/89

PAGE #

TIME

4:41:07 pm

4/10

LINE #

SOURCE TEXT

```
361 {
362     char modeler_reset = reset_modeler_state;
363     u_long err;
364     cpu_control_reg_struct *cpu_ctl = (cpu_control_reg_struct *) CPU_CONTROL_REG;
365
366     /* release for reset
367     */
368     if( reset_modeler_state != REBOOT )
369     {
370         (void)lm_wvarm_access(&modeler_reset, MODELER_RESET, sizeof(MODELER_RESET), MEMORY_WRITE, (u_long *) &err);
371         reset_modeler_state = SHUTDOWN;
372     }
373     (void)lm_wvarm_access(&reset_modeler_state, MODELER_STATE, sizeof(MODELER_STATE), MEMORY_WRITE, (u_long *) &err);
374     cpu_ctl->suicide = 1;
375     for(;;)
376 }
377 }
```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

bsp.diags/modem.c

DATE

5/23/89

PAGE #

TIME

4:41:08 pm

1/11

SOURCE TEXT

```

1  /* SOCS_ID: modem.c rev 3.1, 4/24/89 at 07:55:06 */
2  #include "common.h"
3  #include "cpu_ver.h"
4  #include "in_rd_wr.h"
5  #include "mod_err.h"
6  #include "vvar.h"
7
8  #define Duart_ar      (((int *)0x10c20014))
9  #define Srb           (((int *)0x10c20024))
10 #define Uart_h        (((int *)0x10c2002c))
11
12 #define GLOCK(q)      ((q)->qlock)
13 #define PUTPARK(q)    ((q)->putpark)
14 #define GETPARK(q)    ((q)->getpark)
15 #define ENPTY(q)      ((q)->backp == (q)->frontp)
16
17 #define UART_BUF_SIZE 256
18
19 struct qcontrol {
20     char *qlock,
21     int backp,
22     int frontp,
23     char *getpark,
24     int getwaiters,
25     char *putpark,
26     int putwaiters,
27     char qbuf[UART_BUF_SIZE],
28 };
29
30 struct qcontrol Modem_txq, Modem_rxq;
31
32 static int *Save_duart_inx;
33
34 #define read_duart_status() *((int *)0x10c20004)
35
36 modem_init()
37 {
38     qinit(&Modem_txq);
39     qinit(&Modem_rxq);
40 }
41
42 modemprintline(s)
43 char *s;
44 {
45     while (*s) {
46         qputc(&Modem_txq, *s++);
47     }
48     flushmodem();
49 }
50
51 getmodem()
52 {
53     extern int Uart_mask;
54
55     /* this txq stuff should be in housekeeping task */
56     if (!ENPTY(&Modem_txq)) {
57         Uart_mask |= 0x10000000;
58         Duart_ar = Uart_mask;
59     }
60     return qgetc(&Modem_rxq);
61 }
62
63 getmodem_tmo(tmo, err)
64 int tmo, *err;
65 {
66     extern int Uart_mask;
67
68     /* this txq stuff should be in housekeeping task */
69     if (!ENPTY(&Modem_txq)) {
70         Uart_mask |= 0x10000000;
71         Duart_ar = Uart_mask;
72     }
73     return qgetc_tmo(&Modem_rxq, tmo, err);
74 }
75
76 flushmodem()
77 {
78     if (!ENPTY(&Modem_txq)) {
79         Uart_mask |= 0x10000000;
80         Duart_ar = Uart_mask;
81     }
82 }
83
84 putmodem(c)
85 int c;
86 {
87     qputc(&Modem_txq, c);
88     Uart_mask |= 0x10000000;
89     Duart_ar = Uart_mask;
90 }
91
92 modem_rx()
93 {
94     register int c;
95     register int arb;
96     register struct qcontrol *qp = &Modem_rxq;
97
98     arb = Srb;
99     c = Uart_h >> 24;
100     if (arb & 0x00000000) {
101         /* is this a break? */
102         if (arb & 0x80000000) {
103             Uart_rx_brk();
104         }
105         return;
106     }
107     if (qfull(qp)) return; /* waste character */
108     qp->qbuf[qp->backp++] = c;
109     if (qp->backp >= UART_BUF_SIZE) qp->backp = 0;
110     if (qp->getwaiters != 0) {
111         qp->getwaiters--;
112         ac_post(GETPARK(qp), (char *)1, &err);
113     }
114 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

bsp.diags/modem.c

DATE

5/23/89

PAGE #

TIME

4:41:08 pm

2/12

SOURCE TEXT

```

121 modem_tx()
122 {
123     int err;
124     extern int uart_mask;
125     register struct qcontrol *qp = &modem_c;
126
127     if (!EMPTY(qp)) {
128         uart_b = qp->qbuf(qp->frontp++) << 24;
129         if (qp->frontp >= UART_BUFSIZE) qp->frontp = 0;
130         if (qp->putwaiters != 0) {
131             qp->putwaiters--;
132             ac_post(PUTPARK(qp), (char *)1, &err);
133         }
134         uart_mask |= 0x10000000;
135         uart_ar = uart_mask;
136     } else {
137         uart_mask &= 0x0f000000;
138         uart_ar = uart_mask;
139     }
140 }
141
142 qinit(qp)
143 struct qcontrol *qp;
144 {
145     qp->qlock = (char *)1;
146     qp->backp = 0;
147     qp->frontp = 0;
148     qp->getpark = 0;
149     qp->getwaiters = 0;
150     qp->putpark = 0;
151     qp->putwaiters = 0;
152 }
153
154 qputc(qp, c)
155 struct qcontrol *qp;
156 {
157     int err;
158
159     ac_post(QLOCK(qp), 0, &err);
160     while (qfull(qp)) {
161         ++qp->putwaiters;
162         ac_post(QLOCK(qp), (char *)1, &err);
163         ac_post(PUTPARK(qp), 0, &err);
164         ac_post(QLOCK(qp), 0, &err);
165     }
166     qp->qbuf(qp->backp++) = c;
167     if (qp->backp >= UART_BUFSIZE) qp->backp = 0;
168     if (qp->getwaiters != 0) {
169         qp->getwaiters--;
170         ac_post(GETPARK(qp), (char *)1, &err);
171     }
172     ac_post(QLOCK(qp), (char *)1, &err);
173     if (c == (int)'\n')
174         qputc(qp, (int)'\r'); /* Recursive Cool */
175 }
176
177 qgetc(qp)
178 struct qcontrol *qp;
179 {
180     int c, err;
181
182     ac_post(QLOCK(qp), 0, &err);
183     while (EMPTY(qp)) {
184         ++qp->getwaiters;
185         ac_post(QLOCK(qp), (char *)1, &err);
186         ac_post(GETPARK(qp), 0, &err);
187         ac_post(QLOCK(qp), 0, &err);
188     }
189     c = qp->qbuf(qp->frontp++);
190     if (qp->frontp >= UART_BUFSIZE) qp->frontp = 0;
191     if (qp->putwaiters != 0) {
192         qp->putwaiters--;
193         ac_post(PUTPARK(qp), (char *)1, &err);
194     }
195     ac_post(QLOCK(qp), (char *)1, &err);
196     return c;
197 }
198
199 /* get.c with timeout */
200 qgetc_two(qp, two, err)
201 struct qcontrol *qp;
202 int two, *err;
203 {
204     int c;
205
206     ac_post(QLOCK(qp), two, err);
207     if (*err == 0) return 0;
208     while (EMPTY(qp)) {
209         ++qp->getwaiters;
210         ac_post(QLOCK(qp), (char *)1, err);
211         ac_post(GETPARK(qp), two, err);
212         if (*err == 0) return 0;
213         ac_post(QLOCK(qp), two, err);
214         if (*err == 0) return 0;
215     }
216     c = qp->qbuf(qp->frontp++);
217     if (qp->frontp >= UART_BUFSIZE) qp->frontp = 0;
218     if (qp->putwaiters != 0) {
219         qp->putwaiters--;
220         ac_post(PUTPARK(qp), (char *)1, err);
221     }
222     ac_post(QLOCK(qp), (char *)1, err);
223     return c;
224 }
225
226 qfull(qp)
227 struct qcontrol *qp;
228 {
229     int n;
230     n = qp->frontp - qp->backp - 1;
231     return ((n == 0) || (n == -UART_BUFSIZE));
232 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

tasks.diahs/hkeeping_task.c

DATE

5/23/89

PAGE #

TIME

4:42:31 pm

1/1

```

1  // SCCS ID: hkeeping_task.c rev 3.1, 4/24/89 at 07:55:11
2
3  #include "common.h"
4  #include "cpu.h"
5  #include "network.h"
6  #include "lanes.h"
7  #include "and_err.h"
8  #include "nvars.h"
9
10 unsigned long timer_semaphore;
11 extern u_short check_connection_for_life();
12 extern u_long ln_tick;
13 extern CONNECTION *table_of_conns();
14 extern u_long network_timeout;
15 extern BOOT_STRUCT boot;
16 static char string[ 50 ];
17 #define MAX_TRIES 3
18
19 void
20 housekeeping_task()
21 {
22     cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG,
23     int err;
24     u_char value=0;
25     #ifdef BROKEN_HARDWARE
26     extern char reinitialize_lanec;
27     #endif
28     #ifdef BROKEN_HARDWARE
29     u_char users = 0;
30     register CONNECTION **conn_table;
31     register CONNECTION *conn;
32     /*
33     * This loop is the house keeping task.
34     */
35     while(1)
36     {
37         ac_spin( timer_semaphore, 0, &err);
38         if(err) printf("\nerror in spin in housekeeping tx.", err);
39         if( value == 1)
40         {
41             p_cpu_ctl_reg->net_test_led = not_LED_OFF;
42             value = 0;
43         }
44         else
45         {
46             p_cpu_ctl_reg->net_test_led = not_LED_ON;
47             value = 1;
48         }
49     }
50     #ifdef BROKEN_HARDWARE
51     if( reinitialize_lanec == 1)
52     {
53         reinitialize_lanec = 0;
54         (void)ac_lock();
55         if( get_lane_ready_to_go() != SUCCESS)
56             printf("Can't reinitialize lane\n");
57         else
58             printf("Reinitialized lane\n");
59         if( start_lane() != SUCCESS)
60             printf("Can't start lane\n");
61         else
62             printf("Restarted lane\n");
63         (void)ac_unlock();
64     }
65     #endif
66     /* go thru conn structures
67     * checking for timeouts
68     */
69     conn_table = table_of_conns( 0 );
70     for( users = 0; users < MAX_USERS; users++ )
71     {
72         conn = *conn_table++;
73         CPU_DISABLE_INTERRUPTS;
74         /*
75         * close the connection
76         */
77         if( conn->do_close == TRUE )
78         {
79             if( close_connection_for_server( conn ) == FAILURE ) {
80                 sprintf(string, "Unable to close connection %d\n", conn->fd);
81                 output_routine(string);
82             }
83             else
84             {
85                 /* did we send the last part of the reply
86                 * and is this connection supposed to close
87                 */
88                 if( conn->am_sending == FALSE && conn->am_closing == TRUE ) {
89                     conn->do_close = TRUE;
90                 }
91             }
92         }
93         if( network_timeout != 0 && conn != (CONNECTION *) NULL && conn->am_timing_out == TRUE )
94         {
95             if( conn->time_to_live < (ln_tick * 5) )
96             {
97                 if( check_connection_for_life( conn ) == FAILURE )
98                 {
99                     sprintf(string, "Can't check connection for life in housekeeping task user %d\n", users);
100                     output_routine(string);
101                 }
102                 conn->time_to_live = (ln_tick * 5) + network_timeout;
103                 if( conn->number_of_live_retries > MAX_TRIES )
104                 {
105                     /*
106                     * The host daemon is not responding to daemon
107                     * requests. This does not mean that the host is dead,
108                     * so just make a note of it and continue.
109                     */
110                     sprintf(string, "Warning: Connection timeout for user: %d\n", users);
111                     output_routine(string);
112                     sprintf(string, "Check host daemon.\n");
113                     output_routine(string);
114                     conn->number_of_live_retries = 0;
115                 }
116             }
117         }
118     }
119     CPU_ENABLE_INTERRUPTS;
120 }

```

000615

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
tasks.diags/hkeeping_task.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:42:31 pm | 2/2 |

| LINE # | SOURCE TEXT |
|--------|---|
| 121 | |
| 122 | static void |
| 123 | set_test_led() |
| 124 | { |
| 125 | cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG, |
| 126 | p_cpu_ctl_reg->test_led = test_LED_ON; |
| 127 | } |
| 128 | |
| 129 | static void |
| 130 | clear_test_led() |
| 131 | { |
| 132 | cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG, |
| 133 | p_cpu_ctl_reg->test_led = test_LED_OFF; |
| 134 | } |

| | | | | |
|--|--|--|--------------------|---------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM tasks.diahs/receive_task.c | DATE 5/23/89 | PAGE # 1/3 |
| | | | TIME 4:42:31 pm | |
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCCS ID: receive_task.c rev 3.1, 4/24/89 at 07:55:14 */ | | | |
| 2 | #include <stdio.h> | | | |
| 3 | #include "common.h" | | | |
| 4 | #include "message.h" | | | |
| 5 | #include "task.h" | | | |
| 6 | #include "network.h" | | | |
| 7 | | | | |
| 8 | extern CONNECTION *table_of_conns[], | | | |
| 9 | | | | |
| 10 | void | | | |
| 11 | receive_task() | | | |
| 12 | { | | | |
| 13 | u_char user; | | | |
| 14 | | | | |
| 15 | if (init_socket() == FAILURE) { | | | |
| 16 | printf("Unable to initialize socket\n"); | | | |
| 17 | dequeue_all_messages(); | | | |
| 18 | /* What should we do now? */ | | | |
| 19 | } | | | |
| 20 | | | | |
| 21 | for (;;) { | | | |
| 22 | switch (lm_choose_connection(&user)) { | | | |
| 23 | case PENDING: | | | |
| 24 | if (lm_send_reply(table_of_conns[user]) != SUCCESS) { | | | |
| 25 | printf("user to send reply failed\n", user); | | | |
| 26 | break; | | | |
| 27 | case FAILURE: | | | |
| 28 | dequeue_all_messages(); | | | |
| 29 | break; | | | |
| 30 | } | | | |
| 31 | } | | | |
| 32 | | | | |
| 33 | static | | | |
| 34 | dequeue_all_messages() | | | |
| 35 | { | | | |
| 36 | u_short type; | | | |
| 37 | char str[MAX_MESSAGE]; | | | |
| 38 | | | | |
| 39 | while (lm_dequeue_message(&type, str) != FAILURE) { | | | |
| 40 | if (type == ERROR_MSG) | | | |
| 41 | (void)printf("ERROR: %s\n", str); | | | |
| 42 | else (void)printf("WARNING: %s\n", str); | | | |
| 43 | } | | | |
| 44 | | | | |
| 45 | | | | |
| 46 | | | | |
| 47 | } | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
tasks.diags/serial_task.c

DATE 5/23/89 PAGE #
TIME 4:42:31 pm 1/4

```

1  /* SCCS ID: serial_task.c rev 3.1, 4/24/89 at 07:55:16 */
2  /*
3  ** serial_task
4  */
5  #include "fifo.h"
6  #include "task.h"
7  #include "common.h"
8  #include "netsock.h"
9  #include "lm_diags.h"
10
11 static char Met_diag_message[256];
12
13 void
14 serial_task()
15 {
16     static char buffer;
17     struct fifo_entry fifo;
18
19     fifo.fifo_no = RX_FIFO;
20     fifo.user = 0;
21     fifo.task = SERIAL_TASK_ID;
22
23     for (;;) {
24         fifo.data = (char *)ac_getrc();
25         if (diag_fifo_put(&fifo) == FAILURE) printf("Fifo Failure\n");
26     }
27 }
28
29 void
30 modem_task()
31 {
32     static char buffer;
33     struct fifo_entry fifo;
34
35     fifo.fifo_no = RX_FIFO;
36     fifo.user = 1;
37     fifo.task = SERIAL_TASK_ID;
38
39     for (;;) {
40         fifo.data = (char *)getmodemc();
41         if (diag_fifo_put(&fifo) == FAILURE) printf("Fifo Failure\n");
42     }
43 }
44
45 send_error_message (conn, message)
46     CONNECTION *conn;
47     char *message;
48     long cmd;
49 {
50     cmd = LM_GET_LONG(conn);
51     output_error_message(conn, cmd, message);
52 }
53
54 output_error_message (conn, cmd, message)
55     CONNECTION *conn;
56     char *message;
57 {
58     register long i;
59
60     LM_CHK_PUT_LONG(conn, cmd + 1);
61     LM_CHK_PUT_LONG(conn, 1);
62     LM_CHK_PUT_CHAR(conn, (char) 1);
63     for (i = 0; i < strlen(message); i++)
64         LM_CHK_PUT_CHAR(conn, message[i]);
65     LM_CHK_PUT_CHAR(conn, 0);
66     lm_send_repl(conn);
67 }
68
69 char *
70 diag_get_mode(user, task)
71 {
72     if (task == RECEIVE_TASK_ID)
73         return Met_diag_message;
74     if (user == 0)
75         return "Modem is running diagnostics from the console\n";
76     else
77         return "Modem is running diagnostics over the modem\n";
78 }
79
80 #define LM_PEEK_LONG(X) \
81     (((long *)X)->incoming_buffer_pointer)
82
83 diag_fifo_put(fifo)
84     struct fifo_entry *fifo;
85 {
86     static int User = 0;
87     static int Task = 0;
88     static char diag_username[32] = "unknown user";
89     static char diag_hostname[32] = "unknown host";
90     static char *mode;
91     static char *s, *name_pointer;
92     CONNECTION *conn;
93     int cmd;
94     int i;
95
96     if (Task == 0) {
97         sprintf(Met_diag_message,
98             "%s is running diagnostics over the network from %s\n",
99             diag_username, diag_hostname);
100
101         if (fifo->task == RECEIVE_TASK_ID) {
102             conn = table_of_conns[fifo->user];
103             cmd = LM_PEEK_LONG(conn);
104             if (cmd != LM_DIAG_GINRE) {
105                 mode = "RUNNING DIAGNOSTICS\n";
106                 output_error_message(conn, cmd, mode);
107                 set_close_connection_for_server(conn);
108                 return SUCCESS;
109             }
110             if (4 < ((unsigned long *) (conn->incoming_buffer_pointer + 4)) {
111                 name_pointer = conn->incoming_buffer_pointer + 8;
112                 while (*s++ = *name_pointer++)
113                     ;
114                 s = diag_hostname;
115             }
116         }
117     }
118 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
tasks.diags/serial_task.c

DATE 5/23/89
TIME 4:42:31 pm

PAGE #
2/5

| LINE # | SOURCE TEXT |
|--------|--|
| 121 | while (*p++ == "same_pointer") |
| 122 | |
| 123 | |
| 124 | printf("Net diag message |
| 125 | %s is running diagnostics over the network from %s\n", |
| 126 | diag_username, diag_hostname); |
| 127 | |
| 128 | |
| 129 | User = fifo->user; |
| 130 | Task = fifo->task; |
| 131 | mode = diag_get_mode(User, Task); |
| 132 | printf(mode); |
| 133 | modemprintline(mode); |
| 134 | } |
| 135 | |
| 136 | if ((User == fifo->user) && (Task == fifo->task)) { |
| 137 | return fifo_put(fifo); |
| 138 | } else switch (fifo->task) { |
| 139 | case SERIAL_TASK_ID: |
| 140 | if ((char)fifo->data == INTR_CHARACTER) { |
| 141 | mode = diag_get_mode(fifo->user, fifo->task); |
| 142 | if (Task == RECEIVE_TASK_ID) { |
| 143 | /* close network connection */ |
| 144 | conn = table_of_conns[User]; |
| 145 | cmd = LM_PEEK_LONG(conn); |
| 146 | output_error_message(conn, cmd, mode); |
| 147 | } |
| 148 | User = fifo->user; |
| 149 | Task = fifo->task; |
| 150 | printf(mode); |
| 151 | modemprintline(mode); |
| 152 | return fifo_put(fifo); |
| 153 | } else { |
| 154 | if (fifo->user == 0) { |
| 155 | printf(mode); |
| 156 | } else { |
| 157 | modemprintline(mode); |
| 158 | } |
| 159 | } |
| 160 | break; |
| 161 | case RECEIVE_TASK_ID: |
| 162 | conn = table_of_conns[fifo->user]; |
| 163 | cmd = LM_PEEK_LONG(conn); |
| 164 | output_error_message(conn, cmd, mode); |
| 165 | set_close_connection_for_server(conn); |
| 166 | break; |
| 167 | } |
| 168 | return SUCCESS; |
| 169 | } |

1321

5,353,243

1322

| | | | | | |
|--|---|---|--|--------------------|---------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM tasks.diags/transmit_task.c | | DATE 5/23/89 | PAGE # 1/6 |
| | | | | TIME 4:42:31 pm | |
| LINE # | SOURCE TEXT | | | | |
| 1 | /* SOCS_ID: transmit_task.c Rev: 1.1 4/24/89 at 07:55:19 */ | | | | |
| 2 | void | | | | |
| 3 | transmit_task() | | | | |
| 4 | { | | | | |
| 5 | tx_task(); | | | | |
| 6 | } | | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/delay.c

DATE 5/23/89
TIME 6:14:35 pm

PAGE #
1/1

```

1  /* SCCS_ID: delay.c rev 3.1, 4/24/89 at 07:52:37 */
2  #include "device.h"
3  #include "hardware.h"
4  #include "esprcm.h"
5  #include "lmsrvar.h"
6  #include "protass.h"
7
8  get_delay(def_ptr, instance, ident_change, ident_inconsistent_pins,
9            source_pin_number, source_pin_value)
10 {
11     DEVICE_SPEC *def_ptr;
12     INSTANCE_INFO *instance;
13     PTRN_BITS_LONGWORD *ident_change;
14     PTRN_BITS_LONGWORD *ident_inconsistent_pins;
15     register u_short source_pin_number;
16     u_short source_pin_value;
17
18     PIN_SPEC *pin_def;
19     DAB_INFO *dab_ptr;
20     PIN_INFO *pin_info;
21     MIN_TYP_MAX *min_typ_ptr;
22     MIN_TYP_MAX *old_min_typ_ptr;
23     register TIMING_SPEC *timing_ptr;
24     register u_short max_timing_ptr;
25     register u_short key_state;
26     register u_long ident_change_word;
27     register u_long mask;
28     register char bitno;
29     u_short dest_pin_number;
30     u_short unit_pin_number_offset;
31     u_short word_pin_number_offset;
32     u_char unitno;
33     u_char wordno;
34     dest_pin_value;
35
36     /* If an output pin has several different path delays from the Eval/Store
37      * input pins, then the delay for that output is calculated as shown in the
38      * following table.
39      *
40      * PREVIOUS EVENT      CURRENT EVENT      ACTION
41      *
42      * none                eval table index == -1    table index == -1
43      * none                eval table index == x      table index == x
44      *
45      * table index == -1    eval table index == -1    table index == -1
46      * table index == -1    eval table index == x      table index == x
47      * table index == x      eval table index == -1    table index == x
48      * table index == x      eval table index == y      combine -> comp delay
49      * comp delay == x      eval table index == -1    ? no change
50      * comp delay == x      eval table index == x      combine
51      *
52      * none                store table index == -1    table index == -1
53      * none                store table index == x      table index == x
54      *
55      * table index == -1    store table index == -1    table index == -1
56      * table index == -1    store table index == x      table index == x
57      * table index == x      store table index == -1    table index == -1
58      * table index == x      store table index == x      table index == x
59      * comp delay == x      store table index == -1    table index == -1
60      * comp delay == x      store table index == x      table index == x
61      *
62      * NOTE:
63      * "none" -> this is the first event output
64      * "table index == -1" -> no delay is found in the delay table
65      * "comp delay" -> the delay is a combination of
66      *                 delay table entries
67      * "combine" -> combine the min/typ/max to get
68      *                 worst case delay
69      * "eval" -> this is an eval event
70      * "store" -> this is a store event
71
72     /* ??? Note: make sure that the control field of ident_change is 0.
73      */
74
75     dab_ptr = dab_list[instance->dab_info_index];
76     unit_pin_number_offset = 0;
77     for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
78         word_pin_number_offset = unit_pin_number_offset + 79;
79         for (wordno = 0; wordno < 3; ++wordno) {
80             ident_change_word = ident_change[unitno].word[wordno];
81             for (bitno = 31; bitno >= 0; --bitno) {
82                 mask = bitno_to_mask(bitno);
83                 /* Get out of the "for" loop if there are no more set
84                  * bits to look at.
85                  */
86                 if (ident_change_word == 0)
87                     break;
88                 if (ident_change_word & mask) {
89                     /* reset the bit */
90                     ident_change_word = mask;
91                     dest_pin_number = word_pin_number_offset + bitno - 31;
92                     if (read_ptrn_bit_long(ident_inconsistent_pins[unitno],
93                                           wordno, (u_char)bitno) != 0) {
94                         dest_pin_value = LOGIC_U;
95                     }
96                     else
97                         dest_pin_value =
98                             read_pin_value_long(instance->last_sample_value,
99                                                    unitno, wordno, (u_char)bitno);
100                     pin_info = instance->pin_info_table[dest_pin_number];
101                     pin_def = def_ptr->pin_table[dest_pin_number];
102                     timing_ptr = pin_def->delay_table[0];
103                     max_timing_ptr = timing_ptr->pin_def->delay_cat;
104                     key_state = dest_pin_value << 4 | source_pin_value;
105                     for (; timing_ptr < max_timing_ptr; ++timing_ptr) {

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/delay.c

DATE 5/23/89

PAGE #

TIME 6:14:35 pm

2/2

```

121          if (source_pin_number == timing_ptr->minor_pin &&
122              (u_short)(key_state & ((u_short *)timing_ptr)[1]) ==
123              key_state)
124              break;
125
126      pin_info->new_raw = dest_pin_value;
127      if (pin_info->output_pin_is_linked == FALSE) {
128          pin_info->output_pin_is_linked = TRUE;
129          pin_info->event_pin_number = source_pin_number;
130          pin_info->delay_type = DELAY_TABLE;
131          pin_info->next_output_pin_index =
132              instance->first_data_pin_index;
133          instance->first_data_pin_index = dest_pin_number;
134          if (timing_ptr == max_timing_ptr)
135              pin_info->min_delay = -1;
136          else
137              pin_info->min_delay = timing_ptr->ntym_index;
138      }
139      else {
140          if (timing_ptr != max_timing_ptr) {
141              if (pin_info->delay_type == DELAY_TABLE) {
142                  if (pin_info->min_delay == -1) {
143                      pin_info->min_delay = timing_ptr->ntym_index;
144                      pin_info->event_pin_number = source_pin_number;
145                      pin_info->delay_type = DELAY_TABLE;
146                  }
147                  else {
148                      new_ntym_ptr = &def_ptr->ntym_table[
149                          timing_ptr->ntym_index];
150                      old_ntym_ptr = &def_ptr->ntym_table[
151                          pin_info->min_delay];
152                      combine_delay(&pin_info->min_delay,
153                                  &pin_info->typ_delay,
154                                  &pin_info->max_delay,
155                                  new_ntym_ptr, old_ntym_ptr);
156                      if (source_pin_number < pin_info->event_pin_number)
157                          pin_info->event_pin_number = source_pin_number;
158                      pin_info->delay_type = DELAY_TABLE_COMPOSITE;
159                  }
160              }
161              else {
162                  new_ntym_ptr = &def_ptr->ntym_table[
163                      timing_ptr->ntym_index];
164                  old_ntym_ptr = (MIN_TTP_MAX) * &pin_info->min_delay;
165                  combine_delay(&pin_info->min_delay,
166                              &pin_info->typ_delay,
167                              &pin_info->max_delay,
168                              new_ntym_ptr, old_ntym_ptr);
169                  if (source_pin_number < pin_info->event_pin_number)
170                      pin_info->event_pin_number = source_pin_number;
171                  pin_info->delay_type = DELAY_TABLE_COMPOSITE;
172              }
173          }
174          }
175          }
176          }
177          }
178          }
179          }
180          }
181          }
182          }
183          }
184          }
185          }
186          }
187          }
188          }
189          }
190          }
191          }
192          }
193          }
194          }
195          }
196          }
197          }
198          }
199          }
200          }
201          }
202          }
203          }
204          }
205          }
206          }
207          }
208          }
209          }
210          }
211          }
212          }
213          }
214          }
215          }
216          }
217          }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/edgepl.c

DATE

5/23/89

PAGE #

TIME

6:14:35 pm

1/3

```

LINE # SOURCE TEXT
1  /* SCCS_ID: edgepl.c rev 3.1, 4/24/89 at 07:52:41 */
2  #include "device.h"
3  #include "hardware.h"
4  #include "common.h"
5  #include "lm1000.h"
6  #include "message.h"
7
8  #define NOT_FINISHED -1
9  #define FINISHED -2
10 #define THRESHOLD_FUDGE_FACTOR 4
11 #define def_ptr_def_duty_cycle 50
12
13 place_edges(def_ptr, dab_ptr)
14 DEVICE_SPEC *def_ptr;
15 DAB_INFO *dab_ptr;
16 {
17     EXTRA_DEVICE_SPEC *extra_def_ptr;
18     PIN_SPEC *pin_def;
19     u_long extra_time_needed;
20     u_long sample_min_delay;
21     u_long sample_jitter_error;
22     u_long early_sample_min_delay;
23     u_long early_sample_jitter_error;
24     u_long start_dead_time; /* dead time at the beginning of PCLK period */
25     u_long end_dead_time; /* dead time at the end of PCLK period */
26     u_long actual_end_dead_time;
27     u_long user_hold_time;
28     u_long user_setup_time;
29     u_long hold_time;
30     u_long setup_time;
31     u_long sample_edge_time1;
32     u_long sample_edge_time2;
33     u_long data_edge_time;
34     u_long store_edge_time;
35     u_long hdoif_edge_time;
36     u_long hdoif_edge_time1;
37     u_long clock_jitter_error;
38     u_long edge_jitter_error;
39     u_long quantization_error;
40     u_long physical_clock_period;
41     u_long logical_clock_period;
42     u_long xlogical_clock_period;
43     u_long min_logical_clock_period;
44     u_long max_logical_clock_period;
45     u_long max_modular_logical_clock_period;
46     u_long a_reg;
47     u_long b_reg;
48     u_long source_reg;
49     u_long temp;
50     u_long other_edge_time;
51     u_long constant_time;
52     u_long measured_period;
53     u_long edge_times(MAX_EDGE_COUNT);
54     u_long ret;
55     u_long minimum_fall_sample_time;
56     u_short pin_number;
57     u_char has_store_DWRZ_pins;
58     u_char has_store_R1_or_R2_pins;
59     u_char has_io_store_pins;
60     u_char bits_per_pin = 1; /* This will be declared in DABDEF */
61     u_char loop_count = 0;
62     u_char all_io_store_pins_are_hard_driven;
63     u_char all_io_data_pins_are_hard_driven;
64     u_char any_hard_drives_io_pins;
65     u_char pattern_unit_count;
66     u_char threshold1;
67     u_char threshold2;
68     u_char threshold3;
69     u_char put_hdoif_edge_after_store_edge;
70
71     /* The declared clock period in the DABDEF is the device's clock period.
72     * If the device used DWRZ clock (but so R1/R2 clock) then this period
73     * must be divided by 2. Furthermore, if there are "n" unit-patterns/lane
74     * then this period is further divided by "n".
75     */
76
77     DPRINTF(("inside place_edges\n"));
78     extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
79
80     all_io_store_pins_are_hard_driven = TRUE;
81     all_io_data_pins_are_hard_driven = TRUE;
82     has_store_DWRZ_pins = FALSE;
83     has_store_R1_or_R2_pins = FALSE;
84     has_io_store_pins = FALSE;
85     bits_per_pin = 1;
86     any_hard_drives_io_pins = FALSE;
87     put_hdoif_edge_after_store_edge = FALSE;
88
89     for (pin_number = 0; pin_number < def_ptr->pin_cnt; ++pin_number) {
90         pin_def = def_ptr->pin_table[pin_number];
91         if ((pin_def->direction == NONE) ||
92             (pin_def->direction == POWER) ||
93             (pin_def->direction == CROOND) ||
94             (pin_def->direction == NC))
95             continue;
96
97         if (pin_def->pin_class == STORE) {
98             if (pin_def->clk_format == DWRZ)
99                 has_store_DWRZ_pins = TRUE;
100             else if ((pin_def->clk_format == R1) || (pin_def->clk_format == R2))
101                 has_store_R1_or_R2_pins = TRUE;
102         }
103
104         if (pin_def->direction == IO) {
105             if (pin_def->pin_class == STORE)
106                 has_io_store_pins = TRUE;
107
108             switch (pin_def->in_seq_drive) {
109                 case M_DRIVE:
110                     all_io_store_pins_are_hard_driven = FALSE;
111                 case NO_DRIVE:
112                     all_io_data_pins_are_hard_driven = FALSE;
113                 case R_DRIVE:
114                     any_hard_drives_io_pins = TRUE;
115                 case R2_DRIVE:
116                     any_hard_drives_io_pins = TRUE;
117             }
118         }
119     }
120 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/edgepl.c

DATE 5/23/89

PAGE #

TIME 6:14:35 pm

2/4

```

121         break;
122     default:
123         break;
124     }
125
126     switch (pin_def->last_cyc_drive) {
127     case R_DRIVE:
128     case RM_DRIVE:
129         any_hard_drives_io_pins = TRUE;
130         break;
131     default:
132         break;
133     }
134 }
135
136 pattern_unit_count = bits_per_pin * dab_ptr->unit_count_per_lane;
137
138 /* Measure the clock frequency if we are using external clock */
139 switch (def_ptr->clock_type) {
140 case INTERNAL:
141     /* We only multiply the hardware frequency by 2 if there are no R1/R2
142     * clocks because otherwise the R1/R2 clock will be running too fast.
143     */
144     if ((has_store_DWRZ_pins == TRUE) &&
145         (has_store_R1_or_R2_pins == FALSE)) {
146         min_logical_clock_period = def_ptr->clock_period1 / 2;
147         max_logical_clock_period = def_ptr->clock_period2 / 2;
148     }
149     else {
150         min_logical_clock_period = def_ptr->clock_period1;
151         max_logical_clock_period = def_ptr->clock_period2;
152     }
153
154     if (def_ptr->device_type == PRIVATE) {
155         def_ptr->clock_period1 = PRIVATE_PATTERN_PERIOD;
156         def_ptr->clock_period2 = MAX_MODELER_CLOCK_PERIOD;
157         min_logical_clock_period = PRIVATE_PATTERN_PERIOD;
158         max_logical_clock_period = MAX_MODELER_CLOCK_PERIOD;
159     }
160     break;
161 case EXTL:
162     source_reg = EXTL_REG_VALUE;
163
164     if (!lm_tm_measure_clock(EXTL_REG_VALUE, &measured_period) == FAILURE) {
165         lm_queue_message(ERROR_MSG, "Timing Generator error: measure clock");
166         return(FAILURE);
167     }
168
169     temp = measured_period * pattern_unit_count;
170
171     if ((has_store_DWRZ_pins == TRUE) &&
172         (has_store_R1_or_R2_pins == FALSE)) {
173         temp = temp * 2;
174     }
175
176     if (temp < def_ptr->clock_period1 *
177         (MAX_PERCENT_TOLERANCE / 100)) {
178         lm_queue_message(ERROR_MSG, "The external clock frequency measured (%0.2f MHz) cannot satisfy the maximum device_speed requirement (%0.2f MHz)",
179             (double)1000000.0 / (double)measured_period,
180             (double)1000000.0 / (double)def_ptr->clock_period1);
181         return(FAILURE);
182     }
183
184     if (temp > def_ptr->clock_period2 *
185         (MAX_PERCENT_TOLERANCE / 100)) {
186         lm_queue_message(ERROR_MSG, "The external clock frequency measured (%0.2f MHz) cannot satisfy the minimum device_speed requirement (%0.2f MHz)",
187             (double)1000000.0 / (double)measured_period,
188             (double)1000000.0 / (double)def_ptr->clock_period2);
189         return(FAILURE);
190     }
191
192     def_ptr->clock_period1 = temp;
193     def_ptr->clock_period2 = temp;
194
195     if ((has_store_DWRZ_pins == TRUE) &&
196         (has_store_R1_or_R2_pins == FALSE)) {
197         min_logical_clock_period = def_ptr->clock_period1 / 2;
198         max_logical_clock_period = def_ptr->clock_period2 / 2;
199     }
200     else {
201         min_logical_clock_period = def_ptr->clock_period1;
202         max_logical_clock_period = def_ptr->clock_period2;
203     }
204
205     break;
206 case EXTI:
207     source_reg = EXTI_REG_VALUE;
208
209     if (!lm_tm_measure_clock(EXTI_REG_VALUE, &measured_period) == FAILURE) {
210         lm_queue_message(ERROR_MSG, "Timing Generator error: measure clock");
211         return(FAILURE);
212     }
213
214     temp = measured_period * pattern_unit_count;
215
216     if ((has_store_DWRZ_pins == TRUE) &&
217         (has_store_R1_or_R2_pins == FALSE)) {
218         temp = temp * 2;
219     }
220
221     if (temp < def_ptr->clock_period1 *
222         (MAX_PERCENT_TOLERANCE / 100)) {
223         lm_queue_message(ERROR_MSG, "The external clock frequency measured (%0.2f MHz) cannot satisfy the maximum device_speed requirement (%0.2f MHz)",
224             (double)1000000.0 / (double)measured_period,
225             (double)1000000.0 / (double)def_ptr->clock_period1);
226         return(FAILURE);
227     }
228
229     if (temp > def_ptr->clock_period2 *
230         (MAX_PERCENT_TOLERANCE / 100)) {
231         lm_queue_message(ERROR_MSG, "The external clock frequency measured (%0.2f MHz) cannot satisfy the minimum device_speed requirement (%0.2f MHz)",
232             (double)1000000.0 / (double)measured_period,
233             (double)1000000.0 / (double)def_ptr->clock_period2);
234         return(FAILURE);
235     }
236

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/edgepl.c

DATE 5/23/89
TIME 6:14:35 pm

PAGE #
3/5

```

LINE # SOURCE TEXT
237 }
238
239 def_ptr->clock_period1 = temp;
240 def_ptr->clock_period2 = temp;
241
242 if ((has_store_EWE2_pins == TRUE) &&
243     (has_store_R1_or_R2_pins == FALSE)) {
244     min_logical_clock_period = def_ptr->clock_period1 / 2;
245     max_logical_clock_period = def_ptr->clock_period2 / 2;
246 }
247 else {
248     min_logical_clock_period = def_ptr->clock_period1;
249     max_logical_clock_period = def_ptr->clock_period2;
250 }
251
252 break;
253
254 default:
255     lm_queue_message(ERROR_MSG, "internal error: unknown clock type");
256     return(FAILURE);
257 }
258
259 /* Relax the frequency range by 1 % */
260 min_logical_clock_period = min_logical_clock_period / 100; /*
261
262 DPRINTF(("xedge_skew_time: %d\n", def_ptr->edge_skew_time));
263
264 /* Calculate HOLD time */
265 if (def_ptr->hold_time == -1)
266     hold_time = 15000;
267 else
268     hold_time = def_ptr->hold_time;
269
270 user_hold_time = hold_time;
271
272 DPRINTF(("xraw_hold_time: %d\n", hold_time));
273
274 hold_time += def_ptr->edge_skew_time;
275
276 if (all_io_store_pins_are_hard_driven == TRUE)
277     hold_time += def_ptr->hd_settling_time;
278 else
279     hold_time += def_ptr->md_settling_time;
280
281 DPRINTF(("xneeded_hold_time: %d\n", hold_time));
282
283 /* Calculate SETUP time */
284 if (def_ptr->setup_time == -1)
285     setup_time = 50000;
286 else
287     setup_time = def_ptr->setup_time;
288
289 user_setup_time = setup_time;
290
291 DPRINTF(("xraw_setup_time: %d\n", setup_time));
292
293 setup_time += def_ptr->edge_skew_time;
294
295 if (all_io_data_pins_are_hard_driven == TRUE)
296     setup_time += def_ptr->hd_settling_time;
297 else
298     setup_time += def_ptr->md_settling_time;
299
300 if (def_ptr->device_type == PUBLIC) {
301     if (any_hard_driven_io_pins == TRUE) {
302         /* ULTRA FAST */
303         if (has_io_store_pins == TRUE) {
304             if (setup_time < EDOFF_OFFSET + EDOFF_TO_STORE_OFFSET) {
305                 setup_time = EDOFF_OFFSET + EDOFF_TO_STORE_OFFSET;
306             }
307             if ((hold_time > EDOFF_HOLD_TIME_THRESHOLD) ||
308                 (max_logical_clock_period > EDOFF_PERIOD_THRESHOLD) ||
309                 (hold_time + setup_time > EDOFF_PERIOD_THRESHOLD)) {
310                 put_edoff_edge_after_store_edge = TRUE;
311             }
312         }
313         else {
314             if (setup_time < EDOFF_OFFSET)
315                 setup_time = EDOFF_OFFSET;
316         }
317     }
318
319     DPRINTF(("xneeded_setup_time: %d\n", setup_time));
320
321     if (min_logical_clock_period <
322         MIN_MODELER_CLOCK_PERIOD * pattern_unit_count) {
323         min_logical_clock_period = MIN_MODELER_CLOCK_PERIOD * pattern_unit_count;
324     }
325
326     if (def_ptr->clock_type == INTERNAL) {
327         /* Get the actual clock period achievable by the Timing Generator and
328            also the clock jitter error.
329            */
330         if (lm_tsg_get_frequency_setting(min_logical_clock_period /
331                                         pattern_unit_count,
332                                         &physical_clock_period,
333                                         &clock_jitter_error,
334                                         &u_reg,
335                                         &k_reg,
336                                         &source_reg) == FAILURE) {
337             lm_queue_message(ERROR_MSG, "Timing Generator error: get frequency");
338             return(FAILURE);
339         }
340     }
341     else {
342         /* External clock --> clock jitter = 1% of period */
343         clock_jitter_error = measured_period / 100;
344         u_reg = 0;
345         k_reg = 0;
346         physical_clock_period = measured_period;
347     }
348
349     logical_clock_period = physical_clock_period * pattern_unit_count;
350
351     if (lm_tsg_get_quantization_and_jitter_error(logical_clock_period,
352                                                  &quantization_error,
353                                                  &edge_jitter_error) == FAILURE) {
354         lm_queue_message(ERROR_MSG, "Timing Generator error: get quantization error");
355         return(FAILURE);
356     }

```


Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/edgepl.c

DATE 5/23/89

PAGE #

TIME 6:14:35 pm

4/6

```

LINE # SOURCE TEXT
357
358
359 if (lm_tag_get_dead_time(logical_clock_period,
360 start_dead_time,
361 read_dead_time) == FAILURE) {
362 lm_queue_message(ERROR_MSG, "Timing Generator error: get dead time");
363 return(FAILURE);
364 }
365
366 if (start_dead_time < MAGIC_OUTPUT_DEAD_TIME)
367 start_dead_time = MAGIC_OUTPUT_DEAD_TIME;
368
369 /* The Timing Generator can only generate edges on the first 8 PCLK
370 periods of the logical clock period. If the number of pattern units
371 per line is greater than 8, then consider the remaining PCLK periods
372 as end dead time in the edge time calculation.
373
374 if (pattern_unit_count > 8) {
375 end_dead_time += (pattern_unit_count - 8) * physical_clock_period;
376 }
377
378 if (def_ptr->device_type == PRIVATE) {
379 /* If PRIVATE mode, then increase the dead time by half of the clock
380 period to reserve place to put the DOFF edge.
381 actual_end_dead_time = end_dead_time;
382 end_dead_time += PRIVATE_PATTERN_PERIOD / 2;
383 }
384
385 else {
386 /* For PUBLIC device, increase the dead time by STORE_TO_EDOFF_OFFSET
387 * if put_half_edge_after_store_edge flag is TRUE.
388 if (put_half_edge_after_store_edge == TRUE) {
389 actual_end_dead_time = end_dead_time;
390 end_dead_time += STORE_TO_EDOFF_OFFSET + quantization_error +
391 2 * edge_jitter_error;
392 }
393 }
394
395 if (lm_tag_get_sample_ramp_dead_time(logical_clock_period,
396 (u_long)EDGE7SAMPLETRIGGERMODE,
397 sample_min_delay,
398 sample_jitter_error) == FAILURE) {
399 lm_queue_message(ERROR_MSG, "Timing Generator error: get sample ramp dead time");
400 return(FAILURE);
401 }
402
403 if (lm_tag_get_sample_ramp_dead_time(logical_clock_period,
404 (u_long)EARLYSAMPLETRIGGERMODE,
405 early_sample_min_delay,
406 early_sample_jitter_error) == FAILURE) {
407 lm_queue_message(ERROR_MSG, "Timing Generator error: get sample ramp dead time");
408 return(FAILURE);
409 }
410
411 max_modeler_logical_clock_period =
412 MAX_MODELER_CLOCK_PERIOD * pattern_unit_count;
413
414 DPRINTF(("clock jitter error: %d\n", clock_jitter_error));
415
416 while ((ret = calculate_edge_times(def_ptr,
417 logical_clock_period,
418 sample_min_delay, sample_jitter_error,
419 early_sample_min_delay,
420 edge_jitter_error, quantization_error,
421 start_dead_time, end_dead_time + clock_jitter_error,
422 hold_time + quantization_error + clock_jitter_error +
423 2 * edge_jitter_error,
424 setup_time + quantization_error + 2 * edge_jitter_error,
425 extra_time_needed,
426 sample_edge_time1,
427 sample_edge_time2,
428 data_edge_time,
429 store_edge_time) != FINISHED) {
430
431 if (ret == FAILURE)
432 return(FAILURE);
433
434 /* Return FAILURE if calculate_edge_times() returns NOT FINISHED */
435 if (def_ptr->clock_type != INTERNAL) {
436 lm_queue_message(ERROR_MSG, "the external clock period is too short to meet the setup and hold requirements");
437 return(FAILURE);
438 }
439
440 /* Save guard to avoid infinite loop */
441 --loop_count;
442 if (loop_count == 10) {
443 lm_queue_message(ERROR_MSG, "internal error: cannot place edges in 10 iterations");
444 return(FAILURE);
445 }
446
447 logical_clock_period += extra_time_needed;
448
449 if (logical_clock_period > max_modeler_logical_clock_period) {
450 lm_queue_message(ERROR_MSG, "this device requires clock period greater than maximum modeler clock period (%d ns)",
451 max_modeler_logical_clock_period);
452 return(FAILURE);
453 }
454
455 /* Note: different clock period might have different end_dead_time and
456 start_dead_time because we might have to switch to different
457 edge ramp.
458 */
459
460 if (lm_tag_get_frequency_setting(logical_clock_period /
461 pattern_unit_count,
462 physical_clock_period,
463 clock_jitter_error,
464 lm_reg,
465 lk_reg,
466 (source_reg) == FAILURE) {
467 lm_queue_message(ERROR_MSG, "Timing Generator error: get frequency");
468 return(FAILURE);
469 }
470
471 DPRINTF(("clock jitter error: %d\n", clock_jitter_error));
472
473 logical_clock_period = physical_clock_period * pattern_unit_count;
474
475 if (lm_tag_get_quantization_and_jitter_error(logical_clock_period,
476 quantization_error,
477 edge_jitter_error) == FAILURE) {

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/edgepl.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:35 pm | 5/7 |

```

LINE # SOURCE TEXT
477 lm_queue_message(ERROR_MSG, "Timing Generator error: get quantization error");
478 return(FAILURE);
479 }
480
481 if (lm_tm_get_dead_time == cal_clock_period,
482     rt_dead_time,
483     end_dead_time) == FAILURE) {
484     lm_queue_message(ERROR_MSG, "Timing Generator error: get dead time");
485     return(FAILURE);
486 }
487
488 if (pattern_unit_count > 8) {
489     end_dead_time += (pattern_unit_count - 8) * physical_clock_period;
490 }
491
492 if (def_ptr->device_type == PRIVATE) {
493     /* If PRIVATE mode, then increase the dead time by half of the clock
494      * period to reserve place to put the DOFF edge.
495      */
496     actual_end_dead_time = end_dead_time;
497     end_dead_time += PRIVATE_PATTERN_PERIOD / 2;
498 }
499
500 else {
501     /* For PUBLIC device, increase the dead time by STORE_TO_DOFF_OFFSET
502      * if put_hdoif_edge_after_store_edge flag is TRUE.
503      */
504     if (put_hdoif_edge_after_store_edge == TRUE) {
505         actual_end_dead_time = end_dead_time;
506         end_dead_time += STORE_TO_DOFF_OFFSET + quantization_error +
507             2 * edge_jitter_error;
508     }
509 }
510
511 if (start_dead_time < MAGIC_OUTPUT_DEAD_TIME)
512     start_dead_time = MAGIC_OUTPUT_DEAD_TIME;
513
514 if (lm_tm_get_sample_ramp_dead_time(logical_clock_period,
515     (u_long)EARLYSAMPLETRIGGERMODE,
516     sample_mis_delay,
517     sample_jitter_error) == FAILURE) {
518     lm_queue_message(ERROR_MSG, "Timing Generator error: get sample ramp dead time");
519     return(FAILURE);
520 }
521
522 if (lm_tm_get_sample_ramp_dead_time(logical_clock_period,
523     (u_long)EARLYSAMPLETRIGGERMODE,
524     early_sample_mis_delay,
525     early_sample_jitter_error) == FAILURE) {
526     lm_queue_message(ERROR_MSG, "Timing Generator error: get sample ramp dead time");
527     return(FAILURE);
528 }
529
530 if (def_ptr->device_type == PUBLIC) {
531     if (any_hard_driven_pins == TRUE) {
532         /* Ultra Fast mode has DOFF edge DOFF_OFFSET after data_edge_time */
533         hdoif_edge_time = data_edge_time + DOFF_OFFSET +
534             quantization_error + 2 * edge_jitter_error;
535     }
536     else {
537         /* Non-Ultra Fast mode has DOFF edge at store_edge_time */
538         hdoif_edge_time = store_edge_time +
539             quantization_error + 2 * edge_jitter_error;
540     }
541     if (put_hdoif_edge_after_store_edge == TRUE) {
542         hdoif_edge_time2 = logical_clock_period - actual_end_dead_time;
543     }
544     else {
545         hdoif_edge_time2 = hdoif_edge_time;
546     }
547 }
548
549 else {
550     hdoif_edge_time = logical_clock_period - actual_end_dead_time;
551 }
552
553 /* It's OK to exceed the specified period by 1 %. This is necessary
554  * because the TMC can only return frequency with 1 % tolerance.
555  */
556 max_logical_clock_period += max_logical_clock_period / 100;
557
558 if (logical_clock_period > max_logical_clock_period) {
559     /* Adjust the logical_clock_period reported back to the user
560      * if necessary.
561      */
562     if ((has_store_DWRZ_pins == TRUE) && (has_store_R1_or_R2_pins == FALSE))
563         logical_clock_period = logical_clock_period * 2;
564     else
565         logical_clock_period = logical_clock_period;
566 }
567
568 if ((user_setup_time != 0) && (user_hold_time != 0)) {
569     lm_queue_message(WARNING_MSG, "device_name: ts being modeled with a device_speed of 0.2f MHz to guarantee td ns setup time and td ns
570     hold time; to increase the effective device_speed you must specify a smaller device_setup_time and/or a smaller device_hold_time.");
571     def_ptr->device_name =
572         (double)1000000.0 / ((double)logical_clock_period,
573         user_setup_time / 1000,
574         user_hold_time / 1000);
575 }
576
577 else if ((user_setup_time != 0) && (user_hold_time == 0)) {
578     lm_queue_message(WARNING_MSG, "device_name: ts being modeled with a device_speed of 0.2f MHz to guarantee td ns setup time and td ns
579     hold time; to increase the effective device_speed you must specify a smaller device_setup_time.");
580     def_ptr->device_name =
581         (double)1000000.0 / ((double)logical_clock_period,
582         user_setup_time / 1000,
583         user_hold_time / 1000);
584 }
585
586 else if ((user_setup_time == 0) && (user_hold_time != 0)) {
587     lm_queue_message(WARNING_MSG, "device_name: ts being modeled with a device_speed of 0.2f MHz to guarantee td ns setup time and td ns
588     hold time; to increase the effective device_speed you must specify a smaller device_hold_time.");
589     def_ptr->device_name =
590         (double)1000000.0 / ((double)logical_clock_period,
591         user_setup_time / 1000,
592         user_hold_time / 1000);
593 }
594
595 else if ((user_setup_time == 0) && (user_hold_time == 0)) {
596     lm_queue_message(WARNING_MSG, "device_name: ts being modeled with a device_speed of 0.2f MHz to guarantee td ns setup time and td ns
597     hold time; to increase the effective device_speed you must specify a smaller device_setup_time and/or a smaller device_hold_time.");
598     def_ptr->device_name =
599         (double)1000000.0 / ((double)logical_clock_period,
600         user_setup_time / 1000,
601         user_hold_time / 1000);
602 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/edgepl.c

DATE 5/23/89
TIME 6:14:35 pm

PAGE #
6/8

```

LINE #          SOURCE TEXT
593      user_hold_time / 1000);
594
595      }
596
597      extra_def_ptr->actua...clock_period = physical_clock_period;
598      extra_def_ptr->logics...clock_period = logical_clock_period;
599      extra_def_ptr->sample_edge_time1 = sample_edge_time1;
600      extra_def_ptr->sample_edge_time2 = sample_edge_time2;
601      extra_def_ptr->data_edge_time = data_edge_time;
602      extra_def_ptr->hdoiff_edge_time = hdoiff_edge_time;
603      extra_def_ptr->hdoiff_edge_time2 = hdoiff_edge_time2;
604      extra_def_ptr->store_edge_time = store_edge_time;
605      extra_def_ptr->s_reg = s_reg;
606      extra_def_ptr->k_reg = k_reg;
607      extra_def_ptr->source_reg = source_reg;
608
609      temp = extra_def_ptr->store_edge_time -
610      extra_def_ptr->logical_clock_period * def_ptr_def_duty_cycle / 100;
611
612      if ((long)temp < (long)start_dead_time) {
613          temp = start_dead_time;
614      }
615
616      extra_def_ptr->nos_significant_store_edge_time = temp;
617
618      /* Make sure that the hdoiff edge is in the clock period */
619      if (def_ptr->device_type == PUBLIC) {
620          if (hdoiff_edge_time > store_edge_time) {
621              hdoiff_edge_time = store_edge_time;
622              lm_queue_message(ERROR_MSG, "internal error: HDOFF time > store time");
623              return(FAILURE);
624          }
625      }
626
627      edge_times[0] = hdoiff_edge_time;
628      edge_times[1] = temp;
629      edge_times[2] = data_edge_time;
630      edge_times[3] = store_edge_time;
631      edge_times[4] = temp;
632      edge_times[5] = hdoiff_edge_time2;
633
634      other_edge_time = temp;
635
636      center_time = temp + (store_edge_time - temp) / 2;
637
638      /* When using early sample for R1 or R2 clock, we don't want to
639      * search for the edge all the way back to the min threshold of the
640      * sample ramp because we might miss the pulse. This is due to the
641      * fact that the sample ramp is started before PCCLK. So we will only
642      * search back to the center of the pulse.
643      * This is not a problem with the late sample because the minimum
644      * threshold of the sample ramp will be just before the significant
645      * edge of the R1/R2 clock.
646      */
647      if (extra_def_ptr->store_event_mode == EARLYSAMPLETRIGGERMODE) {
648          if (lm_tmg_get_early_sample_thres(RESOLUTION_05_NS, other_edge_time,
649              threshold1) == FAILURE) {
650              lm_queue_message(ERROR_MSG, "internal error: R1/R2 changes after the fastest sample ramp");
651              return(FAILURE);
652          }
653          if (lm_tmg_get_early_sample_thres(RESOLUTION_05_NS, center_time,
654              threshold2) == FAILURE) {
655              lm_queue_message(ERROR_MSG, "internal error: R1/R2 changes after the fastest sample ramp");
656              return(FAILURE);
657          }
658          if (lm_tmg_get_early_sample_thres(RESOLUTION_05_NS, store_edge_time,
659              threshold3) == FAILURE) {
660              lm_queue_message(ERROR_MSG, "internal error: R1/R2 changes after the fastest sample ramp");
661              return(FAILURE);
662          }
663          DPRINTF(("leading center trailing: %d %d %d -> %d %d %d\n",
664              other_edge_time, center_time, store_edge_time,
665              threshold1, threshold2, threshold3));
666          if (threshold1 >= threshold2) {
667              lm_queue_message(ERROR_MSG, "internal error: center of pulse coincides with the leading edge of R1/R2 clock");
668              return(FAILURE);
669          }
670          if (threshold2 >= threshold3 - THRESHOLD_FUDGE_FACTOR) {
671              lm_queue_message(ERROR_MSG, "internal error: center of pulse coincides with the trailing edge of R1/R2 clock");
672              return(FAILURE);
673          }
674          extra_def_ptr->r1_or_r2_min_threshold = threshold2;
675          DPRINTF(("min threshold for R1 or R2 clock: %d\n", threshold2));
676      }
677
678      if (lm_tmg_get_edge_setting(logical_clock_period, edge_times,
679          extra_def_ptr->edge_setting) == FAILURE) {
680          lm_queue_message(ERROR_MSG, "Timing Generator error: get edge setting");
681          return(FAILURE);
682      }
683
684      if (extra_def_ptr->eval_event_mode == EARLYSAMPLETRIGGERMODE) {
685          extra_def_ptr->edge_setting[6] = 255;
686      }
687      else {
688          if (lm_tmg_get_sample_setting(logical_clock_period,
689              sample_edge_time1,
690              extra_def_ptr->edge_setting[6]) == FAILURE) {
691              lm_queue_message(ERROR_MSG, "Timing Generator error: get sample setting");
692              return(FAILURE);
693          }
694      }
695
696      if (extra_def_ptr->store_event_mode == EARLYSAMPLETRIGGERMODE) {
697          extra_def_ptr->edge_setting[7] = 255;
698      }
699      else {
700          if (lm_tmg_get_sample_setting(logical_clock_period,
701              sample_edge_time2,
702              extra_def_ptr->edge_setting[7]) == FAILURE) {
703              lm_queue_message(ERROR_MSG, "Timing Generator error: get sample setting");
704              return(FAILURE);
705          }
706      }
707
708      extra_def_ptr->edge_setting[8] = extra_def_ptr->edge_setting[6];
709
710
711
712

```

```

LINE # SOURCE TEXT
713
714 /* Set HDOFF edge to max(data_threshold + 1, threshold(datatime + 15)) */
715 if (extra_def_ptr->edge_setting[0] < extra_def_ptr->edge_setting[2] + 1) {
716     extra_def_ptr->edge_setting[0] = extra_def_ptr->edge_setting[2] + 1;
717     extra_def_ptr->edge_setting[5] = extra_def_ptr->edge_setting[2] + 1;
718 }
719
720 /* Make sure data edge and store edge are in different bucket. */
721 if (extra_def_ptr->edge_setting[2] == extra_def_ptr->edge_setting[3]) {
722     la_queue_message(ERROR_MSG, "internal error: same threshold for data edge and store edge");
723     return(FAILURE);
724 }
725
726 /* Set the FALLING SAMPLE reg */
727 if (def_ptr->five_state_sample == 0) {
728     /* The user has not specified the five state sample in DANDEF,
729     * use the greater of (2*logical_clock_period) or 1.6 us.
730     * The 1.6 us sample time is OK whether or not we are doing
731     * timing measurement.
732     */
733     if (any_small_soft_driver(def_ptr) == TRUE) {
734         minimum_fall_sample_time = DEFAULT_FALL_SAMPLE_TIME2;
735     }
736     else {
737         minimum_fall_sample_time = DEFAULT_FALL_SAMPLE_TIME;
738     }
739 }
740
741 if (2 * logical_clock_period > minimum_fall_sample_time) {
742     /* Add 1 logical clock period to the sample time because
743     * the sample time is reference to the beginning of the last ptrn.
744     */
745     if (la_tmg_get_falling_sample_setting(physical_clock_period,
746         (u_long)(3 * logical_clock_period),
747         &temp) != SUCCESS) {
748         la_queue_message(ERROR_MSG, "Timing Generator error: get falling sample setting");
749         return(FAILURE);
750     }
751     DPRINTF(("falling sample --> physical_clock_period: %d, delay: %d, setting: %d\n",
752         physical_clock_period, 3 * logical_clock_period, temp));
753 }
754 else {
755     /* Add 1 logical clock period to the sample time because
756     * the sample time is reference to the beginning of the last ptrn.
757     */
758     minimum_fall_sample_time += logical_clock_period;
759     if (la_tmg_get_falling_sample_setting(physical_clock_period,
760         minimum_fall_sample_time,
761         &temp) != SUCCESS) {
762         la_queue_message(ERROR_MSG, "Timing Generator error: get falling sample setting");
763         return(FAILURE);
764     }
765     DPRINTF(("falling sample --> physical_clock_period: %d, delay: %d, setting: %d\n",
766         physical_clock_period, minimum_fall_sample_time, temp));
767 }
768
769 /* We are going to do timing measurement.
770 * The rising sample will be set at 255 * RESOLUTION_40_WS = 1000 ns.
771 * So the earliest time that the falling sample can happen is
772 * 1000 ns + MIN_SAMPLE_PULSE_WIDTH --> 1000 ns.
773 */
774 if (def_ptr->five_state_sample <
775     MAX_SAMPLE_TIME_RANGE * MIN_SAMPLE_PULSE_WIDTH) {
776     la_queue_message(ERROR_MSG, "default sample time value too small; minimum value: %d ns",
777         (MAX_SAMPLE_TIME_RANGE * MIN_SAMPLE_PULSE_WIDTH) / 1000);
778     return(FAILURE);
779 }
780
781 /* Add 1 logical clock period to the sample time because
782 * the sample time is reference to the beginning of the last ptrn.
783 */
784 if (la_tmg_get_falling_sample_setting(physical_clock_period,
785     def_ptr->five_state_sample + logical_clock_period,
786     &temp) != SUCCESS) {
787     la_queue_message(ERROR_MSG, "Timing Generator error: get falling sample setting");
788     return(FAILURE);
789 }
790
791 DPRINTF(("falling sample --> physical_clock_period: %d, delay: %d, setting: %d\n",
792     physical_clock_period,
793     def_ptr->five_state_sample + logical_clock_period, temp));
794
795 extra_def_ptr->falling_sample_reg = temp;
796
797 DPRINTF(("max_logical_period : %d\n", max_logical_clock_period));
798 DPRINTF(("min_logical_period : %d\n", min_logical_clock_period));
799 DPRINTF(("logical_clk_period : %d\n", logical_clock_period));
800 DPRINTF(("actual_phy_clk_period : %d\n", physical_clock_period));
801 DPRINTF(("sample_edge_eval : %d\n", sample_edge_time1));
802 DPRINTF(("sample_edge_store : %d\n", sample_edge_time2));
803 DPRINTF(("data_edge : %d\n", data_edge_time));
804 DPRINTF(("hdooff_edge : %d\n", hdooff_edge_time));
805 DPRINTF(("hdooff_edge2 : %d\n", hdooff_edge_time2));
806 DPRINTF(("store_edge : %d\n", store_edge_time));
807 DPRINTF(("other_edge : %d\n", other_edge_time));
808 DPRINTF(("start_dead_time : %d\n", start_dead_time));
809 DPRINTF(("end_dead_time : %d\n", end_dead_time));
810 DPRINTF(("sample_min_delay : %d\n", sample_min_delay));
811 DPRINTF(("sample_jitter_error : %d\n", sample_jitter_error));
812 DPRINTF(("early_smpl_min_delay : %d\n", early_sample_min_delay));
813 DPRINTF(("early_smpl_jitter_error : %d\n", early_sample_jitter_error));
814 DPRINTF(("xedge_jitter_error : %d\n", edge_jitter_error));
815 DPRINTF(("xhdo_time : %d\n", data_edge_time - logical_clock_period - store_edge_time));
816
817 DPRINTF(("xsetup_time : %d\n",
818     store_edge_time - data_edge_time));
819 DPRINTF(("xquantization_error : %d\n", quantization_error));
820 DPRINTF(("xclock_jitter_error : %d\n", clock_jitter_error));
821 DPRINTF(("xn_reg : %d\n", a_reg));
822 DPRINTF(("xk_reg : %d\n", k_reg));
823 DPRINTF(("xsource_reg : %d\n", source_reg));
824 DPRINTF(("xedge_0_setting : %d\n", extra_def_ptr->edge_setting[0]));
825 DPRINTF(("xedge_1_setting : %d\n", extra_def_ptr->edge_setting[1]));
826 DPRINTF(("xedge_2_setting : %d\n", extra_def_ptr->edge_setting[2]));
827 DPRINTF(("xedge_3_setting : %d\n", extra_def_ptr->edge_setting[3]));

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/edgepl.c

DATE 5/23/89

PAGE #

TIME 6:14:35 pm

8/10

```

LINE # SOURCE TEXT
833 DPRINTF(("xEdge_4_setting : %d\n", extra_def_ptr->edge_setting[4]));
834 DPRINTF(("xEdge_5_setting : %d\n", extra_def_ptr->edge_setting[5]));
835 DPRINTF(("xEdge_6_setting : %d\n", extra_def_ptr->edge_setting[6]));
836 DPRINTF(("xEdge_7_setting : %d\n", extra_def_ptr->edge_setting[7]));
837 DPRINTF(("xfalling_sample_setting: %d\n",
838         extra_def_ptr->falling_sample_reg));
839 DPRINTF(("exiting place_edges\n"));
840 return(SUCCESS);
841 }
842
843
844 calculate_edge_times(def_ptr,
845                     clock_period,
846                     sample_min_delay, sample_jitter_error,
847                     early_sample_min_delay,
848                     edge_jitter_error, quantization_error,
849                     start_pclk_dead_time, end_pclk_dead_time,
850                     hold_time, setup_time,
851                     extra_time_needed,
852                     sample_edge_time1, sample_edge_time2,
853                     data_edge_time, store_edge_time)
854
855 DEVICE_SPEC *def_ptr,
856 u_long clock_period,
857 u_long sample_min_delay,
858 u_long sample_jitter_error,
859 u_long early_sample_min_delay,
860 u_long edge_jitter_error,
861 u_long quantization_error,
862 u_long start_pclk_dead_time,
863 u_long end_pclk_dead_time,
864 u_long hold_time,
865 u_long setup_time,
866 u_long extra_time_needed,
867 u_long sample_edge_time1,
868 u_long sample_edge_time2,
869 u_long data_edge_time,
870 u_long store_edge_time;
871
872 EXTRA_DEVICE_SPEC *extra_def_ptr,
873 u_long store_edge,
874 u_long data_edge,
875 u_long current_setup_time,
876 u_long sample_ramp_dead_time,
877 u_long edge7_sample_ramp_dead_time;
878
879 DPRINTF(("inside calculate_edge_times\n"));
880 DPRINTF(("clock period : %d\n", clock_period));
881 DPRINTF(("sample_min_delay : %d\n", sample_min_delay));
882 DPRINTF(("sample_jitter_error : %d\n", sample_jitter_error));
883 DPRINTF(("early_sample_min_delay : %d\n", early_sample_min_delay));
884 DPRINTF(("edge_jitter_error : %d\n", edge_jitter_error));
885 DPRINTF(("quantization_error : %d\n", quantization_error));
886 DPRINTF(("start_pclk_dead_time : %d\n", start_pclk_dead_time));
887 DPRINTF(("end_pclk_dead_time : %d\n", end_pclk_dead_time));
888 DPRINTF(("hold_time : %d\n", hold_time));
889 DPRINTF(("setup_time : %d\n", setup_time));
890
891 extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
892
893 edge7_sample_ramp_dead_time = sample_min_delay + sample_jitter_error +
894                               edge_jitter_error + quantization_error;
895
896 if (early_sample_min_delay < sample_min_delay) {
897     /* use EARLY SAMPLE mode */
898     sample_ramp_dead_time = early_sample_min_delay;
899 }
900 else {
901     /* use EDGE 7 SAMPLE mode */
902     sample_ramp_dead_time = sample_min_delay + sample_jitter_error +
903                             edge_jitter_error + quantization_error;
904 }
905
906 if (extra_def_ptr->has_io_store == TRUE) {
907     /* IO store pin can change to 1 at PCLK time, so the hold time
908      * has to be satisfied before the next PCLK.
909     */
910     store_edge = - hold_time;
911
912     if (store_edge > - (long)end_pclk_dead_time)
913         store_edge = - end_pclk_dead_time;
914
915     if (start_pclk_dead_time < sample_ramp_dead_time)
916         data_edge = sample_ramp_dead_time;
917     else
918         data_edge = start_pclk_dead_time;
919 }
920 else {
921     /* Does NOT have I/O store */
922     store_edge = -end_pclk_dead_time;
923
924     data_edge = store_edge + hold_time;
925
926     if (start_pclk_dead_time < sample_ramp_dead_time) {
927         /* sample_ramp_dead_time is the limiting factor */
928         if (data_edge < (long)sample_ramp_dead_time)
929             data_edge = sample_ramp_dead_time;
930     }
931     else {
932         /* start_pclk_dead_time is the limiting factor */
933         if (data_edge < (long)start_pclk_dead_time)
934             data_edge = start_pclk_dead_time;
935     }
936 }
937
938 *store_edge_time = clock_period + store_edge;
939 *data_edge_time = data_edge;
940
941 *sample_edge_time1 = sample_min_delay;
942 *sample_edge_time2 = sample_min_delay;
943
944 if (*data_edge_time >= edge7_sample_ramp_dead_time) {
945     /* Use EDGE7 SAMPLE for EVAL event */
946     extra_def_ptr->eval_event_mode = EDGE7/SAMPLETRIGGERMODE;
947     *sample_edge_time1 = *data_edge_time -
948                         (sample_jitter_error + edge_jitter_error + quantization_error);
949     DPRINTF(("Use edge7 sample for EVAL event\n"));
950 }
951 else {
952     /* Use EARLY SAMPLE for EVAL event */

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/edgepl.c

DATE 5/23/89
TIME 6:14:35 pm

PAGE #
9/11

```

LINE #          SOURCE TEXT
953      extra_def_ptr->eval_event_mode = EARLYSAMPLETRIGGERMODE;
954      DPRINTF(("use early sample for EVAL event\n"));
955      }
956
957      if (*store_edge_time > edge7_sample_ramp_dead_time) {
958          /* Use EDGE7 SAMPLE for STORE event */
959          extra_def_ptr->store_event_mode = EDGE7SAMPLETRIGGERMODE;
960          *sample_edge_time2 = *store_edge_time -
961              (sample_jitter_error + edge_jitter_error + quantization_error);
962          DPRINTF(("use edge7 sample for STORE event\n"));
963      }
964      else {
965          /* Use EARLY SAMPLE for STORE event */
966          extra_def_ptr->store_event_mode = EARLYSAMPLETRIGGERMODE;
967          DPRINTF(("use early sample for STORE event\n"));
968      }
969
970      current_setup_time = *store_edge_time - *data_edge_time;
971
972      DPRINTF(("store edge time      : %d\n", *store_edge_time));
973      DPRINTF(("data edge time2      : %d\n", *data_edge_time));
974      DPRINTF(("sample edge time1 (eval): %d\n", *sample_edge_time1));
975      DPRINTF(("sample edge time2 (store): %d\n", *sample_edge_time2));
976
977      if (current_setup_time < (long)setup_time) {
978          *extra_time_needed = setup_time - current_setup_time;
979          DPRINTF(("extra time      : %d\n", *extra_time_needed));
980          return(NOT_FINISHED);
981      }
982
983      return(FINISHED);
984  }
985
986  any_small_soft_driver(def_ptr)
987  DEVICE_SPEC *def_ptr;
988  {
989      /* If only the weakest soft drive high or low is on then return TRUE */
990
991      PIN_SPEC    *pin_spec_ptr;
992      u_short     pin_count;
993      u_short     pinno;
994
995      pin_count = def_ptr->pin_cnt;
996      pin_spec_ptr = &def_ptr->pin_table[0];
997      for (pinno = 0; pinno < pin_count; ++pinno) {
998          if (pin_spec_ptr->direction != IO) {
999              ++pin_spec_ptr;
1000              continue;
1001          }
1002
1003          if ((pin_spec_ptr->s_drive_low == 1) ||
1004              (pin_spec_ptr->s_drive_hi == 1))
1005              return(TRUE);
1006
1007          ++pin_spec_ptr;
1008      }
1009
1010      return(FALSE);
1011  }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/fault.c

DATE 5/23/89
TIME 6:14:37 pm

PAGE #
1/12

```

1  /* SCCS_ID: fault.c new 3.1, 4/24/89 at 07:52:46 */
2
3  #include "device.h"
4  #include "message.h"
5  #include "hardware.h"
6  #include "eeprom.h"
7  #include "lmserver.h"
8
9  setup_fault_pattern(user, inst_ptr, table_index)
10 USER_INFO user;
11 INSTANCE_INFO inst_ptr;
12 u_short table_index;
13 {
14     DEVICE_SPEC dev_ptr;
15     DAB_INFO dab_ptr;
16     INSTANCE_INFO inst_ptr;
17     LANE_ADDR_INFO lane_info;
18     u_long temp_inst_unit_addr;
19     u_long temp_fault_unit_addr;
20     u_short fault_id_table_index;
21     u_short total_unit;
22     u_char lane;
23     u_char unitno;
24     u_short prev_block_number;
25     u_long block_number;
26     u_long link_table_addr;
27     u_short next_to_last_block_number;
28     u_char temp;
29
30     dev_ptr = inst_ptr->dev_definition;
31
32     dab_ptr = dab_list(inst_ptr->dab_info_index);
33
34     total_unit = dab_ptr->unit_count_per_lane * dab_ptr->lane_count;
35
36     if (new_and_link_instance(user, dev_ptr, "fault",
37         dab_ptr->unit_count,
38         fault_id_table_index,
39         (char)inst_ptr->dab_info_index) == FAILURE) {
40         in_queue_message(ERROR_MSG, "out of memory on modular for instance");
41         return(FAILURE);
42     }
43
44     fault_ptr = user->instances[fault_id_table_index];
45
46     fault_ptr->is_fault = TRUE;
47
48     copy_instance_info(inst_ptr, fault_ptr);
49
50     fault_ptr->disjoint_flag = FALSE;
51
52     /* allocates new block in each lane and store address in VAR_SEQ_ADDR */
53     for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {
54         if (dab_ptr->lane_used[lane]) {
55             if (new_block(&fault_ptr->var_seq_addr[lane], &temp,
56                 PARTIAL_BLOCK, lane) == FAILURE) {
57                 fault_ptr->failed_to_alloc_ptr = TRUE;
58                 in_queue_message(ERROR_MSG, "out of Fast Pattern Memory");
59                 return(FAILURE);
60             }
61         }
62     }
63
64     if (last_pattern_spans_2_blocks(inst_ptr) == TRUE) {
65         /* If the last pattern of the instance spans 2 blocks, this means:
66          * that the last 2 blocks of the instance are NOT feedback blocks.
67          * It also means that the instance has at least 2 blocks.
68          */
69
70         /* find the address of the last common block from the beginning of
71          * the instance pattern sequence.
72          */
73         for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {
74             if (dab_ptr->lane_used[lane]) {
75                 prev_block_number = 0;
76                 block_number = ptob(inst_ptr->seq_start_addr[lane]);
77                 link_table_addr = ptol(inst_ptr->seq_start_addr[lane]);
78
79                 next_to_last_block_number =
80                     ptob(inst_ptr->lane_addr[lane].prev_max_addr - PTRN_ADDR_INC);
81
82                 while (block_number != next_to_last_block_number) {
83                     prev_block_number = block_number;
84                     block_number = read_loc_long((u_long *)link_table_addr) &
85                         BLOCK_NUMBER_MASK;
86                     link_table_addr = btol(lane, block_number);
87                 }
88
89                 if (prev_block_number == 0) {
90                     /* There are only 2 blocks in the instance sequence, so
91                      * the fault sequence has to be disjoint from the instance seq.
92                      * The condition will never happen when the instance has
93                      * feedback sequence, because there would be at least 4 blocks
94                      * in the instance sequence:
95                      * 1- block PREPARE + pre feedback seq
96                      * 2- blocks feedback seq
97                      * 2- blocks post feedback seq + real user pattern seq
98                      * (at least 2 blocks because the last pattern
99                       * spans 2 blocks)
100
101                     Therefore we will never have to worry about allocating
102                     feedback blocks when we are creating faults.
103
104                     fault_ptr->disjoint_flag = TRUE;
105
106                     fault_ptr->seq_start_addr[lane] =
107                         fault_ptr->var_seq_addr[lane];
108
109                     fault_ptr->last_common_block_addr[lane] = 0;
110
111                     fault_ptr->last_common_block_is_feedback = FALSE;
112                 }
113             }
114             else {
115                 fault_ptr->seq_start_addr[lane] =
116                     inst_ptr->seq_start_addr[lane];
117
118                 fault_ptr->last_common_block_addr[lane] =
119                     btol(lane, prev_block_number);
120             }
121         }
122     }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/fault.c

DATE 5/23/89
TIME 6:14:37 pm

PAGE #
2/13

```

121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240

SOURCE TEXT

if (fault_ptr->last_common_block_addr[lane0] ==
    inst_ptr->fb_block_addr[lane0]) {
    fault_ptr->last_common_block_is_feedback = TRUE;
}
else {
    fault_ptr->last_common_block_is_feedback = FALSE;
}

/* copy patterns from the instance next to last block to the
 * fault block addressed by VAR_SEQ_ADDR.
 */
copy_patterns(inst_ptr->lane_addr[lane0].prev_max_addr -
    BLOCK_ADDR_INC,
    fault_ptr->var_seq_addr[lane0],
    PTRN_PER_BLOCK);

lane_info = &fault_ptr->lane_addr[lane0];
lane_info->prev_max_addr =
    fault_ptr->var_seq_addr[lane0] + BLOCK_ADDR_INC;
if (new_block(lane_info->max_addr,
    &temp,
    PTRN_PER_BLOCK,
    lane0) == FAILURE) {
    lm_message(MESSAGE_ERROR_MSG, "out of Fast Pattern Memory");
    fault_ptr->failed_to_alloc_ptrn = TRUE;
    return(FAILURE);
}
set_branch(lane_info->prev_max_addr - PTRN_ADDR_INC,
    lane_info->max_addr);
lane_info->max_addr += BLOCK_ADDR_INC;
copy_patterns(inst_ptr->lane_addr[lane0].max_addr -
    BLOCK_ADDR_INC,
    fault_ptr->lane_addr[lane0].max_addr -
    BLOCK_ADDR_INC,
    PTRN_PER_BLOCK);
}

temp_inst_unit_addr = &inst_ptr->
    unit_addr[inst_ptr->cur_unit_addr_index][0];
temp_fault_unit_addr = &fault_ptr->
    unit_addr[fault_ptr->cur_unit_addr_index][0];
for (unitno = 0; unitno < total_unit; ++unitno) {
    lane0 = dab_ptr->unit_location[unitno].lane_no;
    /* Check if the unit is in the next to last block */
    if ((temp_inst_unit_addr[unitno] ==
        (inst_ptr->lane_addr[lane0].prev_max_addr - BLOCK_ADDR_INC)) &&
        (temp_inst_unit_addr[unitno] <
        (inst_ptr->lane_addr[lane0].prev_max_addr))) {
        temp_fault_unit_addr[unitno] =
            fault_ptr->lane_addr[lane0].prev_max_addr -
            (inst_ptr->lane_addr[lane0].prev_max_addr -
            temp_inst_unit_addr[unitno]);
    }
    else {
        temp_fault_unit_addr[unitno] =
            fault_ptr->lane_addr[lane0].max_addr -
            (inst_ptr->lane_addr[lane0].max_addr -
            temp_inst_unit_addr[unitno]);
    }
    fault_ptr->first_user_ptrn_unit_addr[unitno] =
        temp_fault_unit_addr[unitno];
    if (dab_ptr->unit_location[unitno].last_in_lane)
        fault_ptr->lane_addr[lane0].last_unit_addr =
            temp_fault_unit_addr[unitno];
    fault_ptr->lane_addr[lane0].new_block_addr = 0;
}
else {
    /* The last pattern of the instance is completely contained in the
     * the last instance block.
     */
    for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
        if (!dab_ptr->lane_used[lane0])
            continue;
        if (inst_ptr->lane_addr[lane0].prev_max_addr == 0) {
            /* The instance only has 1 block */
            fault_ptr->disjoint_flag = TRUE;
            fault_ptr->seq_start_addr[lane0] =
                fault_ptr->var_seq_addr[lane0];
            fault_ptr->last_common_block_addr[lane0] = 0;
        }
        else {
            fault_ptr->seq_start_addr[lane0] =
                inst_ptr->seq_start_addr[lane0];
            /* If there are feedback sequence, PREV_MAX_ADDR will point
             * to the end of the feedback blocks. We want to make
             * LAST COMMON BLOCK_ADDR point to the beginning of the
             * feedback blocks.
             */
            if (inst_ptr->lane_addr[lane0].prev_max_addr ==
                (inst_ptr->fb_block_size[lane0] +
                BLOCK_ADDR_INC)) {
                /* PREV_MAX_ADDR points to the end of feedback blocks */
                fault_ptr->last_common_block_addr[lane0] =
                    inst_ptr->fb_block_addr[lane0];
            }
        }
    }
}

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/fault.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:37 pm | 3/14 |

```

LINE #      SOURCE TEXT
241      fault_ptr->last_common_block_is_feedback = TRUE;
242      }
243      else {
244      fault_ptr->last_common_block_addr[lanemo] =
245      inst_ptr->lane_addr[lanemo].prev_max_addr - BLOCK_ADDR_INC,
246      fault_ptr->var_seq_addr[lanemo],
247      PTRN_PCR_BLOCK);
248      fault_ptr->last_common_block_is_feedback = FALSE;
249      }
250
251      copy_pattern(inst_ptr->lane_addr[lanemo].max_addr -
252      BLOCK_ADDR_INC,
253      fault_ptr->var_seq_addr[lanemo],
254      PTRN_PCR_BLOCK);
255
256      /* This is necessary to set the SKO_END bit */
257      fault_ptr->lane_addr[lanemo].prev_max_addr =
258      inst_ptr->lane_addr[lanemo].prev_max_addr;
259
260      fault_ptr->lane_addr[lanemo].max_addr =
261      fault_ptr->var_seq_addr[lanemo] + BLOCK_ADDR_INC;
262      }
263
264      temp_inst_unit_addr = inst_ptr->
265      unit_addr(inst_ptr->cur_unit_addr_index)[0];
266
267      temp_fault_unit_addr = fault_ptr->
268      unit_addr(fault_ptr->cur_unit_addr_index)[0];
269
270      /* Setup the unit address for the fault */
271      for (unitno = 0; unitno < total_unit; ++unitno) {
272
273      lanemo = dab_ptr->unit_location[unitno].lane_no;
274
275      temp_fault_unit_addr[unitno] =
276      fault_ptr->lane_addr[lanemo].max_addr -
277      (inst_ptr->lane_addr[lanemo].max_addr -
278      temp_inst_unit_addr[unitno]);
279
280      fault_ptr->first_user_ptrn_unit_addr[unitno] =
281      temp_fault_unit_addr[unitno];
282
283      if (dab_ptr->unit_location[unitno].last_in_lane)
284      fault_ptr->lane_addr[lanemo].last_unit_addr =
285      temp_fault_unit_addr[unitno];
286
287      fault_ptr->lane_addr[lanemo].new_block_addr = 0;
288      }
289
290      }
291
292      /* Setup the LCTCHDB and LCTCMB addresses.
293      * If the fault is not disjoint from the instance, then
294      * just copy the lcychnb and lcycmb addresses from the instance.
295      * If the fault is disjoint from the instance, then we can see above
296      * that the instance can have at most 2 blocks.
297      */
298
299      if (fault_ptr->disjoint_flag == FALSE) {
300      for (unitno = 0; unitno < total_unit; ++unitno) {
301      fault_ptr->lcychnb_addr[unitno] =
302      inst_ptr->lcychnb_addr[unitno];
303
304      fault_ptr->lcycmb_addr[unitno] =
305      inst_ptr->lcycmb_addr[unitno];
306      }
307      }
308      else {
309      for (unitno = 0; unitno < total_unit; ++unitno) {
310      lanemo = dab_ptr->unit_location[unitno].lane_no;
311
312      /* Check if the instance's lcychnb is in prev block */
313      if ((inst_ptr->lcychnb_addr[unitno] >=
314      (inst_ptr->lane_addr[lanemo].prev_max_addr - BLOCK_ADDR_INC)) &&
315      (inst_ptr->lcychnb_addr[unitno] <
316      (inst_ptr->lane_addr[lanemo].prev_max_addr))) {
317
318      fault_ptr->lcychnb_addr[unitno] =
319      fault_ptr->lane_addr[lanemo].prev_max_addr -
320      (inst_ptr->lane_addr[lanemo].prev_max_addr -
321      inst_ptr->lcychnb_addr[unitno]);
322      }
323      else {
324      fault_ptr->lcychnb_addr[unitno] =
325      fault_ptr->lane_addr[lanemo].max_addr -
326      (inst_ptr->lane_addr[lanemo].max_addr -
327      inst_ptr->lcychnb_addr[unitno]);
328      }
329
330      /* Check if the instance's lcycmb is in prev block */
331      if ((inst_ptr->lcycmb_addr[unitno] >=
332      (inst_ptr->lane_addr[lanemo].prev_max_addr - BLOCK_ADDR_INC)) &&
333      (inst_ptr->lcycmb_addr[unitno] <
334      (inst_ptr->lane_addr[lanemo].prev_max_addr))) {
335
336      fault_ptr->lcycmb_addr[unitno] =
337      fault_ptr->lane_addr[lanemo].prev_max_addr -
338      (inst_ptr->lane_addr[lanemo].prev_max_addr -
339      inst_ptr->lcycmb_addr[unitno]);
340      }
341      else {
342      fault_ptr->lcycmb_addr[unitno] =
343      fault_ptr->lane_addr[lanemo].max_addr -
344      (inst_ptr->lane_addr[lanemo].max_addr -
345      inst_ptr->lcycmb_addr[unitno]);
346      }
347      }
348
349      *table_index = fault_id_table_index;
350
351      return(SUCCESS);
352      }
353
354      modify_instance_pattern_addr(inst_ptr, common_ptrn_count,
355      pattern_count,
356      unit_addr,
357      unit_addr2,
358      max_addr,
359      prev_max_addr)
360

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/fault.c

DATE 5/23/89
TIME 6:14:37 pm

PAGE #
4/15

```

LINE # SOURCE TEXT
361 INSTANCE_INFO *inst_ptr;
362 long common_ptr_count;
363 u_long *pattern_count;
364 u_long unit_addr[];
365 u_long unit_addr2[];
366 u_long max_addr[];
367 u_long prev_max_addr[];
368
369 /* This routine changes the following information from the instance which
370 * which is used by setup_fault_pattern() and saves the information in
371 * the temporary variables:
372 * pattern count
373 * unit_addr[]
374 * unit_addr2[]
375 * prev_max_addr
376 * max_addr
377 */
378
379 DAB_INFO *dab_ptr;
380 u_long max_unit_addr[MAX_LANE_COUNT];
381 link_table_addr;
382 u_long prev_block_number;
383 u_long cur_block_number;
384 u_long block_number;
385 u_long *temp_unit_addr;
386 u_long *temp_unit_addr2;
387 u_char lane_no;
388 u_char unit_no;
389
390 dab_ptr = dab_list(inst_ptr->dab_info_index);
391
392 /* Save/modify pattern count */
393 pattern_count = inst_ptr->pattern_count;
394 inst_ptr->pattern_count = inst_ptr->static_pattern_count +
395 dab_ptr->unit_count_per_lane;
396
397 /* Save instance's unit_addr and previous unit_addr */
398 temp_unit_addr = inst_ptr->unit_addr(inst_ptr->cur_unit_addr_index)[0];
399 temp_unit_addr2 =
400 inst_ptr->unit_addr(inst_ptr->cur_unit_addr_index + 1 & 1)[0];
401
402 for (unit_no = 0; unit_no < MAX_UNIT_COUNT; ++unit_no) {
403
404     unit_addr[unit_no] = temp_unit_addr[unit_no];
405     unit_addr2[unit_no] = temp_unit_addr2[unit_no];
406
407     temp_unit_addr[unit_no] =
408     inst_ptr->first_user_ptr_unit_addr[unit_no];
409 }
410
411 /* Save the max_addr, prev_max_addr */
412 for (lane_no = 0; lane_no < MAX_LANE_COUNT; ++lane_no) {
413
414     max_addr[lane_no] = inst_ptr->lane_addr[lane_no].max_addr;
415     prev_max_addr[lane_no] = inst_ptr->lane_addr[lane_no].prev_max_addr;
416     max_unit_addr[lane_no] = 0;
417 }
418
419 while (common_ptr_count > 0) {
420
421     increment_unit_addr(temp_unit_addr,
422 dab_ptr->unit_count,
423 dab_ptr->unit_count_per_lane);
424
425     inst_ptr->pattern_count += dab_ptr->unit_count_per_lane;
426     --common_ptr_count;
427 }
428
429 /* find the maximum unit_addr */
430 for (unit_no = 0; unit_no < dab_ptr->unit_count; ++unit_no) {
431     lane_no = dab_ptr->unit_location[unit_no].lane_no;
432
433     if (max_unit_addr[lane_no] < temp_unit_addr[unit_no])
434         max_unit_addr[lane_no] = temp_unit_addr[unit_no];
435 }
436
437 /* modify the max_addr, prev_max_addr */
438 for (lane_no = 0; lane_no < MAX_LANE_COUNT; ++lane_no) {
439
440     if (!dab_ptr->lane_used[lane_no])
441         continue;
442
443     inst_ptr->lane_addr[lane_no].max_addr =
444     (max_unit_addr[lane_no] & BLOCK_START_MASK) + BLOCK_ADDR_INC;
445
446     /* search the block before the current block occupied by the last unit */
447     prev_block_number = 0;
448     block_number = ptob(inst_ptr->seq_start_addr[lane_no]);
449     link_table_addr = ptob(inst_ptr->seq_start_addr[lane_no]);
450     cur_block_number = ptob(max_unit_addr[lane_no]);
451
452     while (block_number != cur_block_number) {
453
454         prev_block_number = block_number;
455         block_number = read_loc_long((u_long *)link_table_addr) &
456 BLOCK_NUMBER_MASK;
457         link_table_addr = btob(lane_no, block_number);
458     }
459
460     if (prev_block_number == 0)
461         inst_ptr->lane_addr[lane_no].prev_max_addr = 0;
462     else {
463         inst_ptr->lane_addr[lane_no].prev_max_addr =
464         btob(lane_no, prev_block_number) + BLOCK_ADDR_INC;
465
466         if ((inst_ptr->lane_addr[lane_no].prev_max_addr - BLOCK_ADDR_INC) ==
467             inst_ptr->fb_block_addr[lane_no])
468             inst_ptr->lane_addr[lane_no].prev_max_addr +=
469             (inst_ptr->fb_block_size[lane_no] - 1) * BLOCK_ADDR_INC;
470     }
471 }
472
473
474
475
476
477
478
479
480

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/fault.c

DATE 5/23/89 PAGE #
TIME 6:14:37 pm 5/16

```

LINE # SOURCE TEXT
481 }
482
483 restore_instance_patterns_addr(inst_ptr,
484                               patterns_count,
485                               unit_addr,
486                               unit_addr2,
487                               max_addr,
488                               prev_max_addr)
489
490 INSTANCE_INFO *inst_ptr,
491 u_long patterns_count,
492 u_long unit_addr[1],
493 u_long max_addr[1],
494 u_long prev_max_addr[1],
495
496 /* This routine restores the following information for the instance which
497 * which was saved in the temporary variables:
498 * patterns_count
499 * unit_addr[0]
500 * unit_addr[1]
501 * prev_max_addr
502 * max_addr
503 */
504
505 u_long *temp_unit_addr,
506 u_char laneso,
507 u_char unitno,
508
509 /* Restore patterns_count */
510 inst_ptr->patterns_count = patterns_count,
511
512 /* Restore unit_addr[0] */
513 temp_unit_addr = inst_ptr->unit_addr(inst_ptr->cur_unit_addr_index[0],
514 for (unitno = 0, unitno < MAX_UNIT_COUNT; ++unitno)
515 temp_unit_addr[unitno] = unit_addr[unitno],
516
517 /* Restore unit_addr[1] */
518 temp_unit_addr =
519 inst_ptr->unit_addr(inst_ptr->cur_unit_addr_index + 1 + 1)[0],
520 for (unitno = 0, unitno < MAX_UNIT_COUNT; ++unitno)
521 temp_unit_addr[unitno] = unit_addr2[unitno],
522
523 /* Restore max_addr and prev_max_addr */
524 for (laneso = 0, laneso < MAX_LANE_COUNT; ++laneso) {
525 inst_ptr->lane_addr[laneso].max_addr = max_addr[laneso],
526 inst_ptr->lane_addr[laneso].prev_max_addr = prev_max_addr[laneso],
527 }
528
529 last_patterns_spans_2_blocks(instance)
530
531 INSTANCE_INFO *instance,
532
533 /* Return TRUE if the last pattern of the given instance spans over
534 * the last 2 blocks of the pattern sequence, otherwise return FALSE.
535 * Note that we only have the check 1 lane since the pattern grows
536 * proportionately in each lane.
537 */
538
539
540 DAB_INFO *dab_ptr,
541 u_long block_addr,
542 u_long *temp_unit_addr,
543 u_char laneso,
544 u_char unitno,
545 u_char total_unit,
546
547 dab_ptr = dab_list(instance->dab_info_index),
548 laneso = dab_ptr->unit_location[0].lane_no,
549
550 total_unit = dab_ptr->unit_count_per_lane * dab_ptr->lane_count,
551
552 temp_unit_addr = instance->unit_addr(instance->cur_unit_addr_index[0],
553 block_addr = temp_unit_addr[0] & BLOCK_START_MASK,
554
555 for (unitno = 1, unitno < total_unit; ++unitno) {
556 if (dab_ptr->unit_location[unitno].lane_no == laneso) {
557 if ((temp_unit_addr[unitno] & BLOCK_START_MASK) != block_addr)
558 return(TRUE),
559 }
560 }
561
562 return(FALSE),
563
564
565 copy_instance_info(inst_ptr, fault_ptr)
566 INSTANCE_INFO *inst_ptr,
567 INSTANCE_INFO *fault_ptr,
568
569 /* Copy the following information from the instance data structure to the
570 * fault data structure:
571 * PATTERN_COUNT
572 * STATIC_PATTERN_COUNT
573 * COMMON_PATTERN_COUNT
574 * FB_BLOCK_SIZE
575 * FB_BLOCK_ADDR
576 * PTHN_LOADED
577 * ENDCNS_LOADED
578 * ICHCHDS_LOADED
579 * ICHCHDS_LOADED
580 * SIM_PIN_VALUE
581 * LAST_SAMPLE_VALUE
582 */
583
584 DAB_INFO *dab_ptr,
585 u_char laneso,
586 u_char unitno,
587 u_char total_unit,
588
589 dab_ptr = dab_list(inst_ptr->dab_info_index),
590
591 total_unit = dab_ptr->unit_count,
592
593 fault_ptr->patterns_count = inst_ptr->patterns_count,
594 fault_ptr->static_patterns_count = inst_ptr->static_patterns_count,
595 fault_ptr->common_patterns_count = inst_ptr->patterns_count,
596
597 for (laneso = 0, laneso < MAX_LANE_COUNT; ++laneso) {
598 fault_ptr->fb_block_size[laneso] = inst_ptr->fb_block_size[laneso],
599 fault_ptr->fb_block_addr[laneso] = inst_ptr->fb_block_addr[laneso],
600

```

1355

5,353,243

1356

Copyright 1989
Logic Modeling SystemsSOURCE PROGRAM
lm1000/fault.cDATE 5/23/89
TIME 6:14:37 pmPAGE #
6/17

| LINE # | SOURCE TEXT |
|--------|---|
| 601 | for (unitno = 0; unitno < total_unit; ++unitno) { |
| 602 | fault_ptr->ptrn_loaded[unitno] = inst_ptr->ptrn_loaded[unitno]; |
| 603 | fault_ptr->hndemb_loaded[unitno] = inst_ptr->ptrn_loaded[unitno]; |
| 604 | fault_ptr->lcycmdb_loaded[unitno] = inst_ptr->lcycmdb_loaded[unitno]; |
| 605 | fault_ptr->lcycmdb_loaded[unitno] = 1; ptr->lcycmdb_loaded[unitno]; |
| 606 | fault_ptr->sim_pin_value.data[unitno] = |
| 607 | inst_ptr->sim_pin_value.data[unitno]; |
| 608 | fault_ptr->sim_pin_value.hiz[unitno] = |
| 609 | inst_ptr->sim_pin_value.hiz[unitno]; |
| 610 | fault_ptr->sim_pin_value.unknown[unitno] = |
| 611 | inst_ptr->sim_pin_value.unknown[unitno]; |
| 612 | fault_ptr->sim_pin_value.soft[unitno] = |
| 613 | inst_ptr->sim_pin_value.soft[unitno]; |
| 614 | fault_ptr->last_sample_value.data[unitno] = |
| 615 | inst_ptr->last_sample_value.data[unitno]; |
| 616 | fault_ptr->last_sample_value.hiz[unitno] = |
| 617 | inst_ptr->last_sample_value.hiz[unitno]; |
| 618 | fault_ptr->last_sample_value.hiz[unitno] = |
| 619 | inst_ptr->last_sample_value.hiz[unitno]; |
| 620 | fault_ptr->last_sample_value.unknown[unitno] = |
| 621 | inst_ptr->last_sample_value.unknown[unitno]; |
| 622 | fault_ptr->last_sample_value.soft[unitno] = |
| 623 | inst_ptr->last_sample_value.soft[unitno]; |
| 624 | } |
| 625 | } |
| 626 | } |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89
TIME 6:14:38 pm

PAGE #
1/18

```

LINE # SOURCE TEXT
1  /* SCCS_ID: function.c rev 1.7, 5/9/89 at 19:06:33 */
2  #include "common.h"
3  #include "device.h"
4  #include "message.h"
5  #include "hardware.h"
6  #include "seeprom.h"
7  #include "lmserver.h"
8  #include "protnet.h"
9  #include "lm_sfi.h"
10 #include "lmsnet.h"
11 #include "network.h"
12 #include "mod_err.h"
13 #include "nvaran.h"
14
15 #ifdef MODELER
16 #include "vrx.h"
17 #endif
18
19 extern CONNECTION *table_of_conns[];
20
21 /* MUST initialization routine */
22 extern void lm_init_vars();
23
24 #define MAX_TEMP_BUF_SIZE (3*10*1024)
25 #define TICKS_PER_SECOND 300
26
27 extern char *lmsi_version;
28
29 extern long calculate_pel_count();
30
31 extern u_long total_malloc_size;
32 extern u_long available_malloc_size;
33
34 static u_char modeler_is_locked = FALSE;
35 static char user_with_lock[256];
36
37 void process_begin_session_cmd(user)
38 USER_INFO *user;
39 {
40     char *temp_ptr;
41     char error_string[512];
42     u_short temp;
43     u_short i;
44     char c;
45     version_type sim_version;
46     version_type cnc_version;
47     version_type last_compatible_version;
48
49     DPRINTF(("inside process_begin_session_cmd\n"));
50
51     reset_obuf();
52     lm_put_int(BEGIN_SESSION_ANS);
53
54     /* Version number is encoded in simulator_type */
55     sim_version.version = lm_get_int();
56     cnc_version.version = SOFTWARE_REVISION_NUMBER;
57     last_compatible_version.version = LAST_COMPATIBLE_CNC_VERSION;
58
59     switch (sim_version.field.sim_type) {
60     case (LM_FAULT_SIMULATOR & SIM_TYPE_MASK):
61         user->is_fault_simulator = TRUE;
62         break;
63     case (LM_LOGIC_SIMULATOR & SIM_TYPE_MASK):
64         user->is_fault_simulator = FALSE;
65         break;
66     default:
67         lm_queue_message(ERROR_MSG, "internal simulator error: illegal simulator type");
68         end_queue_message();
69         end_put(user->id);
70         return;
71     }
72
73     /* Check Version Number */
74     if ( (sim_version.field.major > cnc_version.field.major)
75         || (sim_version.field.major < last_compatible_version.field.major)
76         || (sim_version.field.major == last_compatible_version.field.major)
77         && (sim_version.field.minor < last_compatible_version.field.minor)) {
78         if (sim_version.field.major > cnc_version.field.major) {
79             lm_queue_message(ERROR_MSG, "Modeler is booted with Core Modeler Code that is incompatible with host application, reboot Modeler with most recent Core Modeler Code release");
80         }
81         else {
82             lm_queue_message(ERROR_MSG, "Modeler is booted with Core Modeler Code that is incompatible with host application, reboot Modeler with previous Core Modeler Code release");
83         }
84         end_queue_message();
85         end_put(user->id);
86         return;
87     }
88
89     temp_ptr = (char *)LM_GET_ADDR(lm_global.conn_ptr);
90     if (strlen(temp_ptr) > MAX_STRING_LENGTH) {
91         lm_queue_message(ERROR_MSG, "internal simulator error: hostname too long");
92         end_queue_message();
93         end_put(user->id);
94         return;
95     }
96     for (i = 0; (c = lm_get_char()) != '\0'; ++i)
97         user->hostname[i] = c;
98     user->hostname[i] = '\0';
99
100     temp_ptr = (char *)LM_GET_ADDR(lm_global.conn_ptr);
101     if (strlen(temp_ptr) > MAX_STRING_LENGTH) {
102         lm_queue_message(ERROR_MSG, "internal simulator error: username too long");
103         end_queue_message();
104         end_put(user->id);
105         return;
106     }
107     for (i = 0; (c = lm_get_char()) != '\0'; ++i)
108         user->username[i] = c;
109     user->username[i] = '\0';
110
111     if (modeler_is_locked == TRUE) {
112         DPRINTF(("intruder: %s@%s\n", user->username, user->hostname));
113         if (strlen(user->username, user_with_lock) != 0) {
114             lm_queue_message(ERROR_MSG, "Modeler is currently locked by: %s",
115                             user_with_lock);
116             abort_user(user);
117             end_queue_message();
118             end_put(user->id);
119         }
120     }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89

PAGE #

TIME 6:14:38 pm

2/19

```

119
120     if (set_connection_for_server(lm_global_conn_ptr) != SUCCESS) {
121         while (1) {
122             if (lm_queue_message(itamp, error_string) != FAILURE)
123                 break;
124             else
125                 break;
126         }
127     }
128     return;
129 }
130 }
131 end_queue_message();
132 end_put(user->id);
133 return;
134 }
135
136 void process_create_def_cmd(user)
137 USER_INFO *user;
138 {
139     DEVICE_SPEC *def_ptr;
140     PIN_SPEC *pin_ptr;
141     DAB_INFO *dab_ptr;
142     char *buffer_ptr;
143     u_long pin_count_mark;
144     u_long number;
145     u_long units;
146     u_short pin_count;
147     u_short actual_pin_count;
148     u_short pinno;
149     u_short def_id_table_index;
150     u_short wtype_count;
151     u_short errors;
152     u_short warnings;
153     char dab_info_index;
154     int i;
155     char c;
156     DEVICE_SPEC *backend_pass;
157
158     DPRINTF(("inside process_create_def_cmd\n"));
159
160     reset_obuf();
161     lm_put_int(CREATE_DEF_AMS);
162
163     number = lm_get_int();
164     units = lm_get_int();
165
166     if (set_time_scale(user, number, units) == FAILURE) {
167         end_queue_message();
168         end_put(user->id);
169         return;
170     }
171
172     lm_init_vars("HMBL");
173     buffer_ptr = LM_GET_ADDR(lm_global_conn_ptr);
174
175     def_ptr = backend_pass(buffer_ptr);
176     if (def_ptr == NULL) {
177         end_queue_message();
178         end_put(user->id);
179         return;
180     }
181
182     /* Convert all devices to PUBLIC if it is fault simulator. */
183     if (user->is_fault_simulator == TRUE) {
184         def_ptr->device_type = PUBLIC;
185     }
186
187     adjust_delay(user, def_ptr);
188
189     lm_message_types(errors, warnings);
190
191     if (errors) {
192         end_queue_message();
193         end_put(user->id);
194         return;
195     }
196
197     if (new_and_link_definition(user, def_ptr,
198                                def_id_table_index) == FAILURE) {
199         free_device(def_ptr);
200         lm_queue_message(ERROR_MSG, "out of memory on modeler for definition");
201         end_queue_message();
202         end_put(user->id);
203         return;
204     }
205
206     if ((short)(dab_info_index = find_dab(dab_list, def_ptr->device_name,
207                                           (u_char)def_ptr->device_type)) != -1) {
208         if ((long)def_ptr->pin_cnt - 1 >=
209             (long)dab_list[dab_info_index]->unit_count * MAX_PIN_PER_UNIT) {
210             rls_definition(user, def_id_table_index);
211             lm_queue_message(ERROR_MSG, "pin number too large; device is only has %d pins",
212                             def_ptr->device_name);
213             dab_list[dab_info_index]->unit_count * MAX_PIN_PER_UNIT;
214             end_queue_message();
215             end_put(user->id);
216             return;
217         }
218     }
219
220     if (fill_in_extra_data(def_ptr, dab_info_index) == FAILURE) {
221         rls_definition(user, def_id_table_index);
222         lm_queue_message(ERROR_MSG, "out of memory on modeler for definition");
223         end_queue_message();
224         end_put(user->id);
225         return;
226     }
227
228     else {
229         rls_definition(user, def_id_table_index);
230         lm_queue_message(ERROR_MSG, "device name is not found or being used as PRIVATE",
231                         def_ptr->device_name);
232         end_queue_message();
233         end_put(user->id);
234         return;
235     }
236 }
237
238 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/function.c

DATE

5/23/89

PAGE #

TIME

6:14:38 pm

3/20

```

LINE # SOURCE TEXT
239 dab_ptr = dab_list[dab_info_index];
240
241 if (place_edges(def_ptr, dab_ptr) == FAILURE) {
242     ris_definition(user, def_id_table_index);
243     end_queue_message();
244     end_put(user->id);
245     return;
246 }
247
248 if (calc_dab_voltage(def_ptr, dab_list[dab_info_index]) == FAILURE) {
249     ris_definition(user, def_id_table_index);
250     end_queue_message();
251     end_put(user->id);
252     return;
253 }
254
255 if (verify_soft_drive_current(def_ptr,
256     dab_list[dab_info_index]) == FAILURE) {
257     ris_definition(user, def_id_table_index);
258     end_queue_message();
259     end_put(user->id);
260     return;
261 }
262
263 end_queue_message();
264
265 /* write the Definition ID */
266 lm_put_short(def_id_table_index);
267
268 lm_put_char(def_ptr->use_default);
269 lm_put_char(def_ptr->report_miss);
270
271 lm_put_int(def_ptr->default_delay.minimum);
272 lm_put_int(def_ptr->default_delay.typical);
273 lm_put_int(def_ptr->default_delay.maximum);
274
275 /* Return the delay table */
276 mtya_count = def_ptr->mtya_cnt;
277 lm_put_int(mtya_count);
278 for (i = 0; i < mtya_count; ++i) {
279     lm_put_int(def_ptr->mtya_table[i].minimum);
280     lm_put_int(def_ptr->mtya_table[i].typical);
281     lm_put_int(def_ptr->mtya_table[i].maximum);
282 }
283
284 /* write pin count */
285 pin_count_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
286 lm_put_int(0);
287
288 /* write pin id's */
289 actual_pin_count = 0;
290 pin_count = def_ptr->pin_cnt;
291 pin_ptr = def_ptr->pin_table;
292 for (pinno = 0; pinno < pin_count; ++pinno) {
293     if (pin_ptr->direction == NONE) {
294         ++pin_ptr;
295         continue;
296     }
297 }
298
299 #ifdef DEBUG
300 DPRINTF((" Pin name: %15s  PN: %3d  ",
301     pin_ptr->pin_name, pinno));
302
303 switch (pin_ptr->pin_class) {
304 case DATA:
305     DPRINTF(("D  "));
306     break;
307 case EVAL:
308     DPRINTF(("E  "));
309     break;
310 case STORE:
311     switch (pin_ptr->clk_format) {
312     case DWZ:
313         DPRINTF(("S  "));
314         break;
315     case R1:
316         DPRINTF(("R  "));
317         break;
318     case R0:
319         DPRINTF(("F  "));
320         break;
321     default:
322         DPRINTF(("S? "));
323         break;
324     }
325     break;
326 default:
327     DPRINTF(("?  "));
328     break;
329 }
330
331 switch (pin_ptr->direction) {
332 case IN:
333     DPRINTF(("I  \n"));
334     break;
335 case OUT:
336     DPRINTF(("O  \n"));
337     break;
338 case IO:
339     DPRINTF(("IO \n"));
340     break;
341 case POWER:
342     DPRINTF(("PWR \n"));
343     break;
344 case GROUND:
345     DPRINTF(("GND \n"));
346     break;
347 case NC:
348     DPRINTF(("NC  \n"));
349     break;
350 default:
351     DPRINTF(("?? \n"));
352     break;
353 }
354 #endif
355
356 ++actual_pin_count;
357
358 if ((pin_ptr->direction == POWER) ||

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/function.c

DATE 5/23/89

PAGE #

TIME 6:14:38 pm

4/21

```

LINE # SOURCE TEXT
359 (pin_ptr->direction == GROUND) ||
360 (pin_ptr->direction == NC)) {
361   lm_put_short(pname);
362   lm_put_short(DATA);
363   lm_put_short(pin_ptr->direction);
364
365   for (i = 0; c = pin_ptr->pin_name[i]; ++i)
366     lm_put_char(c);
367   lm_put_char('\0');
368
369   for (i = 0; c = pin_ptr->pin_number[i]; ++i)
370     lm_put_char(c);
371   lm_put_char('\0');
372
373   if (pin_ptr->pin_alias == NULL) {
374     lm_put_char('\0');
375   }
376   else {
377     for (i = 0; c = pin_ptr->pin_alias[i]; ++i)
378       lm_put_char(c);
379     lm_put_char('\0');
380   }
381
382   ++pin_ptr;
383   continue;
384 }
385
386 lm_put_short(pname);
387 switch (pin_ptr->pin_class) {
388   case DATA:
389     lm_put_short(pin_ptr->pin_class);
390     break;
391   case STORE:
392     /* Figure out whether it's store both, store rise, or store fall */
393     switch (pin_ptr->clk_format) {
394       case DWRZ:
395         lm_put_short(pin_ptr->pin_class);
396         break;
397       case R1:
398         lm_put_short(EDGE_RISE);
399         break;
400       case F1:
401         lm_put_short(EDGE_FALL);
402         break;
403       default:
404         break;
405     }
406     break;
407   default:
408     break;
409 }
410 lm_put_short(pin_ptr->direction);
411
412 for (i = 0; c = pin_ptr->pin_name[i]; ++i)
413   lm_put_char(c);
414 lm_put_char('\0');
415
416 for (i = 0; c = pin_ptr->pin_number[i]; ++i)
417   lm_put_char(c);
418 lm_put_char('\0');
419
420 if (pin_ptr->pin_alias == NULL) {
421   lm_put_char('\0');
422 }
423 else {
424   for (i = 0; c = pin_ptr->pin_alias[i]; ++i)
425     lm_put_char(c);
426   lm_put_char('\0');
427 }
428 ++pin_ptr;
429 }
430
431 LM_PUT_LONG_AT_MARK(pin_count_mark, lm_global_cons_ptr, actual_pin_count);
432
433 end_put(user->fd);
434
435 /* ARGSUSED */
436 void process_check_dabdef_cmd(user)
437 USER_INFO *user;
438 {
439   DEVICE_SPEC *backend_pass();
440   DEVICE_SPEC *def_ptr;
441   char *buffer_ptr;
442
443   DPRINTF(("inside process_check_dabdef_cmd\n"));
444   reset_obuf();
445   lm_put_int(CHECK_DABDEF_ANS);
446
447   lm_init_vars("HGBL");
448   buffer_ptr = LM_GET_ADDR(lm_global_cons_ptr);
449   def_ptr = backend_pass(buffer_ptr);
450
451   if (def_ptr != NULL)
452     free_device(def_ptr);
453
454   end_queue_message();
455   end_put(user->fd);
456 }
457
458 void process_create_instance_cmd(user)
459 USER_INFO *user;
460 {
461   EXTRA_DEVICE_SPEC *extra_def_ptr;
462   DEVICE_SPEC *def_ptr;
463   DAB_INFO *dab_ptr;
464   INSTANCE_INFO *instance;
465   long def_id;
466   char *name_ptr;
467   u_short error_count;
468   u_short warning_count;
469   u_short inst_id_table_index;
470   char dab_info_index;

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:38 pm | 5/22 |

```

LINE # SOURCE TEXT
479
480 DPRINTF(("inside process_create_instance_cmd\n"));
481
482 reset_obuf();
483 lm_put_inst(CREATE_INSTANCE_ANS);
484
485 def_id = lm_get_short();
486
487 name_ptr = LM_CFT_ADDR(lm_global_conn_ptr);
488
489 /* verify the def_id */
490 if ((def_id < 0) || (def_id >= user->def_table_size)) {
491     lm_queue_message(ERROR_MSG, "internal simulator error: invalid definition id: id specified",
492         def_id);
493     end_queue_message();
494     end_put(user->id);
495     return;
496 }
497
498 def_ptr = user->definition[def_id];
499 if (BOGUS_DEFINITION(user, def_ptr)) {
500     lm_queue_message(ERROR_MSG, "internal simulator error: invalid definition id: id specified",
501         def_id);
502     end_queue_message();
503     end_put(user->id);
504     return;
505 }
506
507 extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
508 if (extra_def_ptr->dab_ok == FALSE) {
509     lm_queue_message(ERROR_MSG, "Device Adapter was removed");
510     end_queue_message();
511     end_put(user->id);
512     return;
513 }
514
515 if ((short)(dab_info_index = find_dab(dab_list, def_ptr->device_name,
516     (u_char)def_ptr->device_type)) == -1) {
517     /* If def_ptr->device_type == PUBLIC, then this error can happen if
518      * - someone unplugged the DAB after the definition is created.
519      *
520      * If def_ptr->device_type == PRIVATE, then this error can happen if:
521      * - someone else has grabbed the device after the definition is
522      *   created
523      * - this user has more instances than DAB's.
524      */
525     lm_queue_message(ERROR_MSG, "cannot find device_name: %s",
526         def_ptr->device_name);
527     end_queue_message();
528     end_put(user->id);
529     return;
530 }
531
532 dab_ptr = dab_list[dab_info_index];
533
534 if (def_ptr->device_type == PRIVATE) {
535     dab_ptr->used_as_private = TRUE;
536 }
537
538 if (new_and_link_instance(user, def_ptr, name_ptr,
539     dab_ptr->unit_count,
540     inst_id_table_index,
541     dab_info_index) == FAILURE) {
542     lm_queue_message(ERROR_MSG, "out of memory on modeler for instance");
543     end_queue_message();
544     end_put(user->id);
545     return;
546 }
547
548 instance = user->instance[inst_id_table_index];
549
550 instance->is_fault = FALSE;
551
552 if (def_ptr->device_type == PRIVATE) {
553     set_private_mode(dab_ptr, TRUE);
554 }
555
556 build_static_patterns_seq(instance);
557
558 lm_message_types(&error_count, &warning_count);
559
560 if (error_count != 0) {
561     /* If there are any errors then release this instance since the host
562      * SFI is not going to create the instance if there are errors.
563      */
564     if (def_ptr->device_type == PRIVATE) {
565         set_private_mode(dab_ptr, FALSE);
566     }
567
568     dab_ptr->used_as_private = FALSE;
569     return_all_ptrs_block(instance);
570     rls_instance(user, inst_id_table_index);
571     end_queue_message();
572 }
573
574 else {
575     if (def_ptr->device_type == PRIVATE) {
576         instance->has_history = TRUE;
577         instance->purge_ptrs_on_next_eval = TRUE;
578     }
579
580     dab_ptr->act_inst_count += 1;
581
582     turn_on_in_use(dab_ptr);
583
584     end_queue_message();
585
586     lm_put_short(inst_id_table_index);
587
588     copy_initial_values(instance);
589
590     end_put(user->id);
591 }
592
593 void process_create_fault_cmd(user)
594 USER_INFO user;
595
596 {

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89 PAGE #
TIME 6:14:38 pm 6/23

```

LINE # SOURCE TEXT
599 INSTANCE_INFO *inst_ptr;
600 INSTANCE_INFO *fault_ptr;
601 DAB_INFO *dab_ptr;
602 long inst_id;
603 common_ptrn_count;
604 u_long temp_patterns_count;
605 u_long temp_unit_addr[MAX_UNIT_COUNT];
606 u_long temp_unit_addr2[MAX_UNIT_COUNT];
607 u_long temp_max_addr[MAX_LANE_COUNT];
608 u_long temp_prev_max_addr[MAX_LANE_COUNT];
609 u_short error_count;
610 u_short warning_count;
611 u_short table_index;
612
613 DPRINTF(("inside process_create_fault_cmd\n"));
614
615 reset_obuf();
616 lm_put_inst(CREATE_FAULT_MSG);
617
618 inst_id = lm_get_short();
619
620 /* verify the inst_id */
621 if ((inst_id < 0) || (inst_id > user->inst_table_size)) {
622     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: id specified",
623         inst_id);
624     end_queue_message();
625     end_put(user->id);
626     return;
627 }
628
629 inst_ptr = user->instance(inst_id);
630 if (BEGUS_INSTANCE(user, inst_ptr)) {
631     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: id specified",
632         inst_id);
633     end_queue_message();
634     end_put(user->id);
635     return;
636 }
637
638 if (inst_ptr->definition->device_type == PRIVATE) {
639     /* This condition can happen if the user says it's LOGIC_SIMULATOR
640     * and then call lm_create_fault(). If the simulator is FAULT_SIMULATOR
641     * then we are going to convert the device_type to PUBLIC when
642     * we create definitions.
643     */
644     lm_queue_message(ERROR_MSG, "internal simulator error: cannot create fault on a private mode device");
645     end_queue_message();
646     end_put(user->id);
647     return;
648 }
649
650 if (((EXTRA_DEVICE_SPEC *)
651     inst_ptr->definition->extra_data->dab_ok == FALSE)) {
652     lm_queue_message(ERROR_MSG, "Device Adapter was removed");
653     end_queue_message();
654     end_put(user->id);
655     return;
656 }
657
658 dab_ptr = dab_list[inst_ptr->dab_info_index];
659 common_ptrn_count = lm_get_inst();
660
661 if (common_ptrn_count == -1) {
662     /* The fault patterns should branch out from whatever patterns t1
663     * instance has currently.
664     */
665     (void)setup_fault_patterns(user, inst_ptr, table_index);
666 }
667 else {
668     /* Make setup_fault_patterns() believe that the instance only has
669     * "common_ptrn_count" number of patterns. Save the actual
670     * information about the instance patterns in temp_*.
671     */
672     modify_instance_patterns_addr(inst_ptr, common_ptrn_count,
673         &temp_patterns_count,
674         &temp_unit_addr,
675         &temp_unit_addr2,
676         &temp_max_addr,
677         &temp_prev_max_addr);
678     (void)setup_fault_patterns(user, inst_ptr, table_index);
679     restore_instance_patterns_addr(inst_ptr,
680         &temp_patterns_count,
681         &temp_unit_addr,
682         &temp_unit_addr2,
683         &temp_max_addr,
684         &temp_prev_max_addr);
685 }
686
687 lm_message_types(&error_count, &warning_count);
688
689 if (error_count != 0) {
690     /* If there are any errors then release this instance since the host
691     * SRI is not going to create the instance if there are errors.
692     */
693     fault_ptr = user->instance(table_index);
694     return_all_ptrn_b(fault_ptr);
695     release_instance(user, table_index);
696     end_queue_message();
697 }
698 else {
699     ++inst_ptr->fault_count;
700     dab_ptr->act_var_count += 1;
701     turn_on_in_use(dab_ptr);
702     end_queue_message();
703     lm_put_short(table_index);
704 }
705
706 end_put(user->id);
707
708 void process_release_def_cmd(user)
709 USER_INFO *user;
710 {
711     DEVICE_SPEC *def_ptr;

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:38 pm | 7/24 |

```

LINE # SOURCE TEXT
719 INSTANCE_INFO *instance;
720 long def_id;
721 u_short i;
722
723 DPRINTF(("inside process_release_def_cmd\n"));
724
725 reset_obuf();
726 lm_put_int(RELEASE_DEF_ANS);
727
728 def_id = lm_get_short();
729
730 /* verify the def_id */
731 if ((def_id < 0) || (def_id >= user->def_table_size)) {
732     lm_queue_message(ERROR_MSG, "internal simulator error: invalid definition id: %d specified",
733                     def_id);
734     end_queue_message();
735     end_put(user->id);
736     return;
737 }
738
739 def_ptr = user->definition[def_id];
740 if (BOGUS_DEFINITION(user, def_ptr)) {
741     lm_queue_message(ERROR_MSG, "internal simulator error: invalid definition id: %d specified",
742                     def_id);
743     end_queue_message();
744     end_put(user->id);
745     return;
746 }
747
748 /* Check if this definition still has instances/faults */
749 for (i = 0; i < user->inst_table_size; ++i) {
750     instance = user->instance[i];
751
752     if (BOGUS_INSTANCE(user, instance))
753         continue;
754
755     if (instance->definition == def_ptr) {
756         lm_queue_message(ERROR_MSG, "internal simulator error: Definition id: %d still has instances/faults associated with it",
757                         def_id);
758         end_queue_message();
759         end_put(user->id);
760         return;
761     }
762 }
763
764 rls_definition(user, (u_short)def_id);
765
766 end_queue_message();
767 end_put(user->id);
768 }
769
770 void process_release_instance_cmd(user)
771 USER_INFO *user;
772 {
773     INSTANCE_INFO *instance;
774     DAB_INFO *dab_ptr;
775     long inst_id;
776
777     DPRINTF(("inside process_release_instance_cmd\n"));
778
779     reset_obuf();
780     lm_put_int(RELEASE_INSTANCE_ANS);
781
782     inst_id = lm_get_short();
783
784     /* verify the inst_id */
785     if ((inst_id < 0) || (inst_id >= user->inst_table_size)) {
786         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: %d specified",
787                         inst_id);
788         end_queue_message();
789         end_put(user->id);
790         return;
791     }
792
793     instance = user->instance[inst_id];
794     if (BOGUS_INSTANCE(user, instance)) {
795         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: %d specified",
796                         inst_id);
797         end_queue_message();
798         end_put(user->id);
799         return;
800     }
801
802     if (instance->is_fault) {
803         lm_queue_message(ERROR_MSG, "internal simulator error: instance id: %d is a fault",
804                         inst_id);
805         end_queue_message();
806         end_put(user->id);
807         return;
808     }
809
810     /* Check if this instance still has faults */
811     /* ??? Can't do this with the current structure.
812     * Just make sure this condition is checked on the host.
813     */
814
815     return_all_ptrs_block(instance);
816
817     dab_ptr = dab_list[instance->dab_info_index];
818
819     rls_instance(user, (u_short)inst_id);
820
821     --dab_ptr->act_inst_count;
822
823     if (dab_ptr->used_as_private == TRUE) {
824         dab_ptr->used_as_private = FALSE;
825         set_private_mode(dab_ptr, FALSE);
826     }
827
828     if ((dab_ptr->act_inst_count + dab_ptr->act_var_count) == 0)
829         turn_off_in_use(dab_ptr);
830
831     end_queue_message();
832     end_put(user->id);
833 }
834
835 void process_release_fault_cmd(user)

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89
TIME 6:14:38 pm

PAGE #
8/25

```

LINE # SOURCE TEXT
839 USER_INFO *user;
840 {
841     INSTANCE_INFO *fault;
842     INSTANCE_INFO *instance;
843     DAB_INFO *dab_ptr;
844     long fault_id;
845     u_short inst_id;
846     u_short i;
847
848     DPRINTF(("inside process_release_fault_cmd\n"));
849
850     reset_obuf();
851     lm_put_inst(RELEASE_FAULT_ANS);
852
853     fault_id = lm_get_short();
854     inst_id = lm_get_short();
855
856     /* verify the fault_id */
857     if ((fault_id < 0) || (fault_id >= user->inst_table_size)) {
858         lm_queue_message(ERROR_MSG, "internal simulator error: invalid fault id: %d specified",
859             fault_id);
860         end_queue_message();
861         end_put(user->fd);
862         return;
863     }
864
865     /* verify the inst_id */
866     if ((inst_id < 0) || (inst_id >= user->inst_table_size)) {
867         lm_queue_message(ERROR_MSG, "internal error: invalid inst id: %d specified",
868             inst_id);
869         end_queue_message();
870         end_put(user->fd);
871         return;
872     }
873
874     fault = user->instance[fault_id];
875     if (BOGUS_INSTANCE(user, fault)) {
876         lm_queue_message(ERROR_MSG, "internal simulator error: invalid fault id: %d specified",
877             fault_id);
878         end_queue_message();
879         end_put(user->fd);
880         return;
881     }
882
883     if (! fault->is_fault) {
884         lm_queue_message(ERROR_MSG, "internal simulator error: fault id: %d is an instance",
885             fault_id);
886         end_queue_message();
887         end_put(user->fd);
888         return;
889     }
890
891     instance = user->instance[inst_id];
892     if (BOGUS_INSTANCE(user, instance)) {
893         lm_queue_message(ERROR_MSG, "internal error: invalid inst id: %d specified",
894             inst_id);
895         end_queue_message();
896         end_put(user->fd);
897         return;
898     }
899
900     if (instance->is_fault) {
901         lm_queue_message(ERROR_MSG, "internal error: instance id: %d is a fault",
902             inst_id);
903         end_queue_message();
904         end_put(user->fd);
905         return;
906     }
907
908     return_all_ptrn_block(fault);
909
910     dab_ptr = dab_list[fault->dab_info_index];
911
912     --instance->fault_count;
913
914     rls_instance(user, (u_short) fault_id);
915
916     --dab_ptr->act_var_count;
917
918     if ((dab_ptr->act_inst_count + dab_ptr->act_var_count) == 0)
919         turn_off_in_use(dab_ptr);
920
921     end_queue_message();
922     end_put(user->fd);
923 }
924
925 void process_eval_cmd(user)
926 USER_INFO *user;
927 {
928     INSTANCE_INFO *instance;
929     DEVICE_SPEC *def_ptr;
930     u_short inst_id;
931     u_short error_count;
932     u_short warning_count;
933     u_char replay_count;
934     u_char changed_dac;
935
936     DPRINTF(("inside process_eval_cmd\n"));
937
938     reset_obuf();
939     lm_put_inst(EVAL_ANS);
940
941     inst_id = lm_get_short();
942
943     /* verify inst_id */
944     if (inst_id >= user->inst_table_size) {
945         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
946             inst_id);
947         end_queue_message();
948         end_put(user->fd);
949         return;
950     }
951
952     instance = user->instance[inst_id];
953     if (BOGUS_INSTANCE(user, instance)) {
954         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
955             inst_id);
956         end_queue_message();
957         end_put(user->fd);
958         return;

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89
TIME 6:14:38 pm

PAGE #
10/27

```

1079
1080     else {
1081         (void)evaluate_1_bit_per_pin(def_ptr, instance,
1082                                     gbl_ident_inconsistent_pins,
1083                                     gbl_temp_steady_state_result,
1084                                     gbl_ident_change,
1085                                     replay_count,
1086                                     &changed_dac);
1087     }
1088 }
1089
1090 }
1091
1092 lm_message_types(&error_count, &warning_count);
1093
1094 if (error_count != 0) {
1095     end_queue_message();
1096     end_put(user->id);
1097     return;
1098 }
1099
1100 run_output_bcode(instance);
1101
1102 end_queue_message();
1103
1104 copy_out_pin_changes(instance);
1105
1106 end_put(user->id);
1107 }
1108
1109 void process_save_def_cmd(user)
1110 USER_INFO *user;
1111 {
1112     DEVICE_SPEC *def_ptr;
1113     INSTANCE_INFO *instance;
1114     char *def_buf;
1115     u_long temp;
1116     u_short def_id;
1117     long
1118 #ifdef DEBDC
1119     char
1120 #endif
1121     temp_ptr;
1122
1123     extern char *extract_device();
1124
1125     DPRINTF(("inside process_save_def_cmd\n"));
1126
1127     reset_obuf();
1128     lm_put_inst(SAVE_DEF_ANS);
1129
1130     def_id = lm_get_short();
1131
1132     /* verify def_id */
1133     if (def_id >= user->def_table_size) {
1134         lm_queue_message(ERROR_MSG, "internal simulator error: invalid definition id: %d specified",
1135                         def_id);
1136         end_queue_message();
1137         end_put(user->id);
1138         return;
1139     }
1140
1141     def_ptr = user->definition[def_id];
1142     if (BOCUS_DEFINITION(user, def_ptr)) {
1143         lm_queue_message(ERROR_MSG, "internal simulator error: invalid definition id: %d specified",
1144                         def_id);
1145         end_queue_message();
1146         end_put(user->id);
1147         return;
1148     }
1149
1150     if (user->save_buffer != NULL) {
1151         lm_queue_message(ERROR_MSG, "internal error: left over text from previous save");
1152         end_queue_message();
1153         end_put(user->id);
1154         return;
1155     }
1156
1157     unadjust_delay(user, def_ptr);
1158
1159     def_buf = extract_device(def_ptr);
1160     if (def_buf == NULL) {
1161         lm_queue_message(ERROR_MSG, "internal error: extract_device failed");
1162         end_queue_message();
1163         end_put(user->id);
1164         return;
1165     }
1166
1167     adjust_delay(user, def_ptr);
1168
1169     DPRINTF(("length of extract_device buffer: %d\n", strlen(def_buf)));
1170
1171     #ifdef DEBDC
1172     #endif
1173     temp_ptr = def_buf;
1174     while (*temp_ptr)
1175         ac_putc(*temp_ptr++);
1176     ac_putc('\n');
1177     temp_ptr = def_buf;
1178     while (*temp_ptr)
1179         DPRINTF((" %c", *temp_ptr++));
1180     DPRINTF((" \n"));
1181 #endif
1182 #endif
1183
1184     end_queue_message();
1185
1186     temp = 0;
1187     for (i = 0; i < user->inst_table_size; ++i) {
1188         instance = user->instance[i];
1189         if (BOCUS_INSTANCE(user, instance))
1190             continue;
1191
1192         if (instance->definition != def_ptr)
1193             continue;
1194
1195         if (instance->is_fault == FALSE)
1196             temp += instance->pattern_count;
1197         else
1198             temp += instance->pattern_count -

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89 PAGE #
TIME 6:14:38 pm 11/28

```

1199      instance->common_patterns_count;
1200  }
1201
1202  lm_put_int(temp); /* device pattern count */
1203  lm_put_int(strlen(def_buf)); /* total size of the def_buf */
1204
1205  for (i = 0; def_buf[i] != '\0' && i < SAVE_REST_PTRN_BUFFER_LIMIT; ++i) {
1206      lm_put_char(def_buf[i]);
1207  }
1208  lm_put_char('\0');
1209
1210  if (i < SAVE_REST_PTRN_BUFFER_LIMIT) {
1211      /* The whole definition text fits into the network buffer */
1212      user->save_buffer = NULL;
1213      user->save_buffer_offset = -1;
1214      DPRINTF(def_buf);
1215  }
1216  else {
1217      /* There are more to come */
1218      user->save_buffer = def_buf;
1219      user->save_buffer_offset = i;
1220  }
1221
1222  end_put(user->fd);
1223  }
1224
1225  void process_save_def_cont_cmd(user)
1226  USER_INFO *user;
1227  {
1228      char *def_buf;
1229      long i;
1230      long limit;
1231
1232      DPRINTF(("inside process_save_def_cont_cmd\n"));
1233
1234      reset_obuf();
1235      lm_put_int(SAVE_DEF_CONT_ANS);
1236
1237      if (user->save_buffer == NULL) {
1238          lm_queue_message(ERROR_MSG, "internal error: no more definition text to return");
1239          end_queue_message();
1240          end_put(user->fd);
1241          return;
1242      }
1243      end_queue_message();
1244
1245      def_buf = user->save_buffer;
1246
1247      limit = user->save_buffer_offset + SAVE_REST_PTRN_BUFFER_LIMIT;
1248      for (i = user->save_buffer_offset; def_buf[i] != '\0' && i < limit; ++i) {
1249          lm_put_char(def_buf[i]);
1250      }
1251      lm_put_char('\0');
1252
1253      if (i < limit) {
1254          user->save_buffer = NULL;
1255          user->save_buffer_offset = -1;
1256          DPRINTF(def_buf);
1257      }
1258      else {
1259          user->save_buffer_offset = i;
1260      }
1261      end_put(user->fd);
1262  }
1263
1264  void process_save_ptrn_cmd(user)
1265  USER_INFO *user;
1266  {
1267      INSTANCE_INFO *inst_ptr;
1268      DEVICE_SPEC *def_ptr;
1269      PIN_SPEC *pin_def;
1270      PIN_INFO *pin;
1271      DAB_INFO *dab_ptr;
1272      u_long patterns_count;
1273      u_short inst_id;
1274      u_long pin_count_mark;
1275      u_long patterns_count_follows_mark;
1276      u_long out_buffer_remaining_size;
1277      u_short pinno;
1278      u_short pin_count;
1279      u_char device_ptrn_size; /* size of device pattern is bytes */
1280
1281      DPRINTF(("inside process_save_ptrn_cmd\n"));
1282
1283      user->save_state = SENT_RECV_NOTHING;
1284
1285      reset_obuf();
1286      lm_put_int(SAVE_PTRN_ANS);
1287
1288      inst_id = lm_get_short();
1289
1290      /* verify inst_id */
1291      if (inst_id >= user->inst_table_size) {
1292          lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1293                          inst_id);
1294          end_queue_message();
1295          end_put(user->fd);
1296          return;
1297      }
1298
1299      inst_ptr = user->instance[inst_id];
1300      if (BOGUS_INSTANCE(user, inst_ptr)) {
1301          lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1302                          inst_id);
1303          end_queue_message();
1304          end_put(user->fd);
1305          return;
1306      }
1307
1308      dab_ptr = dab_list[inst_ptr->dab_info_index];
1309
1310      if (inst_ptr->is_fault == FALSE) {
1311          /* This is an instance */
1312
1313          /* The patterns_count includes the last pattern which is a variable
1314             * pattern. This variable pattern is the same as
1315             * INSTANCE_INFO.ptrn_loaded, so we don't have to store this in the
1316             * save set file. Therefore we need to subtract it by
1317             * unit_count_per_line.
1318             */

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89
TIME 6:14:38 pm

PAGE #
12/29

```

1319 user->pattern_to_send_count = inst_ptr->pattern_count -
1320 inst_ptr->static_pattern_count -
1321 dab_ptr->unit_count_per_lane;
1322
1323 }
1324 else {
1325     /* This is a fault */
1326     /* We don't need to subtract it by unit_count_per_lane since both
1327      * pattern_count and common_pattern_count includes this variable
1328      * pattern already.
1329     */
1330     user->pattern_to_send_count = inst_ptr->pattern_count -
1331     inst_ptr->common_pattern_count;
1332 }
1333 end_queue_message();
1334 lm_put_char(dab_ptr->unit_count);
1335
1336 /* The following is a total DEVICE ptrn count, and pattern_to_send_count
1337  * is the number of lane patterns. Therefore we need to divide it by
1338  * unit_count_per_lane.
1339  */
1340 lm_put_int((user->pattern_to_send_count / dab_ptr->unit_count_per_lane), );
1341
1342 /* write common_ptrn_count */
1343 if (inst_ptr->is_fault == FALSE)
1344     lm_put_int(-1);
1345 else {
1346     /* subtract by unit_count_per_lane because common_ptrn_count includes
1347      * the last pattern which is a variable pattern.
1348     */
1349     lm_put_int((inst_ptr->common_pattern_count -
1350     inst_ptr->static_pattern_count -
1351     dab_ptr->unit_count_per_lane) /
1352     dab_ptr->unit_count_per_lane);
1353 }
1354
1355 lm_put_char(inst_ptr->check_input_x_count);
1356 lm_put_char(inst_ptr->first_eval);
1357 lm_put_char(inst_ptr->sample_count);
1358 lm_put_int(inst_ptr->evaluation_count);
1359 lm_put_char(inst_ptr->purge_ptrn_on_next_eval);
1360 lm_put_char(inst_ptr->has_history);
1361 lm_put_char(inst_ptr->use_2_bit_per_pin);
1362 lm_put_char(inst_ptr->enable_timing_mask);
1363 lm_put_char(inst_ptr->had_shorted_pin);
1364
1365 pin_count_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
1366 lm_put_short(0); /* dummy pin_count */
1367
1368 def_ptr = inst_ptr->definition;
1369 pin_count = 0;
1370 for (pinno = 0; pinno < def_ptr->pin_cnt; ++pinno) {
1371     pin_def = &def_ptr->pin_table[pinno];
1372
1373     if ((pin_def->direction == NONE) ||
1374         (pin_def->direction == POWER) ||
1375         (pin_def->direction == GROUND) ||
1376         (pin_def->direction == NC))
1377         continue;
1378
1379     ++pin_count;
1380     pin = &inst_ptr->pin_info_table[pinno];
1381
1382     lm_put_short(pinno);
1383     lm_put_char(pin->old_raw);
1384     lm_put_char(pin->old_filtered);
1385     lm_put_char(pin->new_raw);
1386     lm_put_char(pin->new_filtered);
1387     lm_put_int((pin->pin_time));
1388     lm_put_char(pin->uninitialized_pin);
1389 }
1390
1391 LM_PUT_SHORT_AT_MARK(pin_count_mark, lm_global_conn_ptr, pin_count);
1392
1393 pattern_count_follows_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
1394 lm_put_int(0); /* dummy pattern_count_follows */
1395
1396 out_buffer_remaining_size = SAVE_REST_PTRN_BUFFER_LIMIT;
1397 device_ptrn_size = dab_ptr->unit_count * sizeof(PTRN_BITS);
1398
1399 pattern_count = 0;
1400 while (out_buffer_remaining_size > device_ptrn_size) {
1401     if (send_pattern(user, inst_ptr) == FALSE)
1402         break;
1403     ++pattern_count;
1404     out_buffer_remaining_size -= device_ptrn_size;
1405 }
1406
1407 LM_PUT_LONG_AT_MARK(pattern_count_follows_mark, lm_global_conn_ptr,
1408 pattern_count);
1409
1410 DPRINTF(("pattern_count_follows: %d\n", pattern_count));
1411 end_put(user->fd);
1412 }
1413
1414 void process_save_ptrn_count_cmd(user)
1415 USER_INFO *user;
1416 {
1417     INSTANCE_INFO *inst_ptr;
1418     DAB_INFO *dab_ptr;
1419     u_long pattern_count;
1420     u_short inst_id;
1421     u_long pattern_count_follows_mark;
1422     u_long out_buffer_remaining_size;
1423     u_char device_ptrn_size; /* size of device pattern is bytes */
1424
1425     DPRINTF(("inside process_save_ptrn_count_cmd\n"));
1426
1427     reset_obuf();
1428     lm_put_int(SAVE_PTRN_COUNT_ANS);
1429
1430     inst_id = lm_get_short();
1431
1432     /* verify inst_id */
1433     if (inst_id >= user->inst_table_size) {

```


Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/function.c

DATE

5/23/89

PAGE #

TIME

6:14:38 pm

13/30

```

1439     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1440                       inst_id);
1441     end_queue_message();
1442     end_put(user->id);
1443     return;
1444 }
1445
1446 inst_ptr = user->instance[inst_id];
1447 if (BOGUS_INSTANCE(user, inst_ptr)) {
1448     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1449                       inst_id);
1450     end_queue_message();
1451     end_put(user->id);
1452     return;
1453 }
1454
1455 dab_ptr = dab_list[inst_ptr->dab_info_index];
1456
1457 end_queue_message();
1458
1459 pattern_count_follows_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
1460 lm_put_int(0); /* dummy count */
1461
1462 out_buffer_remaining_size = SAVE_REST_PTRN_BUFFER_LIMIT;
1463
1464 device_ptrn_size = dab_ptr->unit_count * sizeof(PTRN_BITS);
1465
1466 pattern_count = 0;
1467 while (out_buffer_remaining_size > device_ptrn_size) {
1468     if (send_patterns(user, inst_ptr) == FALSE)
1469         break;
1470     ++pattern_count;
1471     out_buffer_remaining_size -= device_ptrn_size;
1472 }
1473
1474 LM_PUT_LONG_AT_MARK(pattern_count_follows_mark, lm_global_conn_ptr,
1475                     pattern_count);
1476
1477 DPRINTF(("pattern_count_follows: %d\n", pattern_count));
1478
1479 end_put(user->id);
1480 }
1481
1482 void process_restore_inst_cmd(user)
1483 USER_INFO *user;
1484 {
1485     INSTANCE_INFO *instance;
1486     PIN_INFO *pin;
1487     u_short inst_id;
1488     u_short pin_count;
1489     u_short pin_number;
1490
1491     DPRINTF(("inside process_restore_inst_cmd\n"));
1492
1493     reset_cbuf();
1494     lm_put_int(RESTORE_INST_ANS);
1495
1496     inst_id = lm_get_short();
1497
1498     /* verify inst_id */
1499     if (inst_id >= user->inst_table_size) {
1500         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1501                           inst_id);
1502         end_queue_message();
1503         end_put(user->id);
1504         return;
1505     }
1506
1507     instance = user->instance[inst_id];
1508     if (BOGUS_INSTANCE(user, instance)) {
1509         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1510                           inst_id);
1511         end_queue_message();
1512         end_put(user->id);
1513         return;
1514     }
1515
1516     instance->check_input_t_count = lm_get_char();
1517     instance->first_eval = lm_get_char();
1518     instance->sample_count = lm_get_char();
1519     instance->evaluation_count = lm_get_int();
1520     instance->purge_ptrn_on_next_eval = lm_get_char();
1521     instance->has_history = lm_get_char();
1522     instance->use_2_bit_per_pin = lm_get_char();
1523     instance->enable_timing_meas = lm_get_char();
1524     instance->had_shorted_pin = lm_get_char();
1525
1526     pin_count = lm_get_short();
1527
1528     for (i = 0; i < pin_count; ++i) {
1529         pin_number = lm_get_short();
1530         pin = instance->pin_info_table[pin_number];
1531         pin->old_raw = lm_get_char();
1532         pin->old_filtered = lm_get_char();
1533         pin->new_raw = lm_get_char();
1534         pin->new_filtered = lm_get_char();
1535         pin->pin_time = lm_get_int();
1536         pin->uninitialized_pin = lm_get_char();
1537     }
1538
1539     end_queue_message();
1540     end_put(user->id);
1541 }
1542
1543 void process_restore_ptrn_cmd(user)
1544 USER_INFO *user;
1545 {
1546     INSTANCE_INFO *instance;
1547     DAB_INFO *dab_ptr;
1548     u_short pattern_count_follows;
1549     u_short inst_id;
1550     u_char unit_count;
1551     u_short i;
1552
1553     DPRINTF(("inside process_restore_ptrn_cmd\n"));
1554 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89
TIME 6:14:38 pm

PAGE #
14/31

```

1559  reset_objf();
1560  lm_put_inst(MEMORY_PTRM_AMS);
1561
1562  inst_id = lm_get_short();
1563  unit_count = lm_get_inst();
1564  pattern_count_follows = lm_get_inst();
1565
1566  /* verify inst_id */
1567  if (inst_id >= user->inst_table_size) {
1568      lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: id specified",
1569                      inst_id);
1570      end_queue_message();
1571      end_put(user->id);
1572      return;
1573  }
1574
1575  instance = user->instance(inst_id);
1576  if (BOGUS_INSTANCE(user, instance)) {
1577      lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: id specified",
1578                      inst_id);
1579      end_queue_message();
1580      end_put(user->id);
1581      return;
1582  }
1583
1584  dab_ptr = dab_list(instance->dab_info_index);
1585
1586  if (dab_ptr->unit_count != unit_count) {
1587      /* ??? error */
1588      lm_queue_message(ERROR_MSG, "internal error: current unit count does not equal saved unit count");
1589      end_queue_message();
1590      end_put(user->id);
1591      return;
1592  }
1593
1594  if (instance->failed_to_alloc_ptrn == TRUE) {
1595      lm_queue_message(ERROR_MSG, "out of Fast Pattern Memory, cannot restore");
1596      end_queue_message();
1597      end_put(user->id);
1598      return;
1599  }
1600
1601  if (pattern_count_follows == 0) {
1602      instance->restore_state = SENT_RECV_NOTHING;
1603
1604      /* write the variable pattern */
1605      write_patterns(instance);
1606      instance->unit_addr(instance->cur_unit_addr_index)[0] =
1607          instance->ptrn_loaded;
1608      end_queue_message();
1609      end_put(user->id);
1610      return;
1611  }
1612
1613  for (i = 0; i < pattern_count_follows; ++i) {
1614      if (restore_inst_pattern(instance, unit_count) == FAILURE)
1615          break;
1616  }
1617
1618  end_queue_message();
1619  end_put(user->id);
1620
1621  void process_ptrn_hist_cmd(user)
1622  USER_INFO *user;
1623
1624  {
1625      INSTANCE_INFO *instance;
1626      u_short inst_id;
1627      u_char command;
1628
1629      DPRINTF(("inside process_ptrn_hist_cmd\n"));
1630
1631      reset_objf();
1632      lm_put_inst(PTRN_HIST_AMS);
1633
1634      inst_id = lm_get_short();
1635
1636      /* verify inst_id */
1637      if (inst_id >= user->inst_table_size) {
1638          lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: id specified",
1639                          inst_id);
1640          end_queue_message();
1641          end_put(user->id);
1642          return;
1643      }
1644
1645      instance = user->instance(inst_id);
1646      if (BOGUS_INSTANCE(user, instance)) {
1647          lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: id specified",
1648                          inst_id);
1649          end_queue_message();
1650          end_put(user->id);
1651          return;
1652      }
1653
1654      if (instance->definition->device_type == PUBLIC) {
1655          end_queue_message();
1656          end_put(user->id);
1657          return;
1658      }
1659
1660      switch (command = lm_get_char()) {
1661      case LM_KEEP_PATTERNS:
1662          if (instance->has_history == TRUE)
1663              instance->purge_ptrn_on_next_eval = FALSE;
1664          else
1665              lm_queue_message(ERROR_MSG, "internal simulator error: instance id: id does not have history",
1666                              inst_id);
1667          break;
1668      case LM_DONT_KEEP_PATTERNS:
1669          if (instance->has_history == TRUE) {
1670              instance->purge_ptrn_on_next_eval = TRUE;
1671              if (instance->enable_timing_meas == TRUE) {
1672                  lm_queue_message(WARNING_MSG, "timing measurement is turned off on private instance: id because pattern history were deleted",
1673                                  instance->device_info_string);
1674              }
1675          }
1676      }
1677  }
1678  
```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:38 pm | 15/32 |

```

1679     }
1680     break;
1681 default:
1682     lm_queue_message(ERROR_MSG, "internal simulator error: illegal command: %d",
1683     lm_queue_message(command));
1684     break;
1685 }
1686
1687 end_queue_message();
1688 end_put(user->id);
1689 }
1690
1691 void process_lm_modeler_cmd(user)
1692 USER_INFO *user;
1693 {
1694     u_long    temp;
1695     u_long    value;
1696     u_short   attrib;
1697     u_char    lane0;
1698     u_char    panno;
1699     u_char    slotno;
1700     u_char    i;
1701
1702     DPRINTF(("inside process_lm_modeler_cmd\n"));
1703
1704     resetobuf();
1705     lm_put_int(INQ_MODELER_ANS);
1706
1707     switch (attrib = lm_get_int()) {
1708     case LM_TOTAL_PATTERNS:
1709         end_queue_message();
1710         temp = 0;
1711         for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0)
1712             for (panno = 0; panno < MAX_PAN_COUNT; ++panno)
1713                 if (system_config->lane[lane0]->pan[panno] != NULL)
1714                     temp += system_config->lane[lane0]->pan[panno]->pan_size;
1715         lm_put_int(temp);
1716         break;
1717     case LM_TOTAL_MODELER_MEMORY:
1718         end_queue_message();
1719         lm_put_int(total_malloc_size);
1720         break;
1721     case LM_AVAILABLE_PATTERNS:
1722         end_queue_message();
1723         temp = count_avail_patterns((u_char)MAX_LANE_COUNT) * PTEN_PER_BLOCK;
1724         lm_put_int(temp);
1725         break;
1726     case LM_AVAILABLE_MODELER_MEMORY:
1727         end_queue_message();
1728         lm_put_int(available_malloc_size);
1729         break;
1730     case LM_NUMBER_OF_USERS:
1731         end_queue_message();
1732         temp = 0;
1733         for (i = 0; i < MAX_USER_COUNT; ++i)
1734             if (user_info_array[i]->active == TRUE)
1735                 ++temp;
1736         lm_put_int(temp);
1737         break;
1738     case LM_WHICH_USER_AM_I:
1739         end_queue_message();
1740         for (i = 0; i < MAX_USER_COUNT; ++i)
1741             if (user_info_array[i] == user)
1742                 break;
1743         lm_put_int(i);
1744         break;
1745     case LM_NUMBER_OF_LANES:
1746         end_queue_message();
1747         lm_put_int(system_config->phy_lane_count);
1748         break;
1749     case LM_NUMBER_OF_SLOTS_PER_LANE:
1750         end_queue_message();
1751         lm_put_int((long)system_config->slot_count);
1752         break;
1753     case LM_NUMBER_OF_PACS:
1754         end_queue_message();
1755         temp = 0;
1756         for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0)
1757             if (system_config->lane[lane0]->pac_present == TRUE)
1758                 ++temp;
1759         lm_put_int(temp);
1760         break;
1761     case LM_NUMBER_OF_PANS:
1762         end_queue_message();
1763         temp = 0;
1764         for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0)
1765             for (panno = 0; panno < MAX_PAN_COUNT; ++panno)
1766                 if (system_config->lane[lane0]->pan[panno] != NULL)
1767                     ++temp;
1768         lm_put_int(temp);
1769         break;
1770     case LM_NUMBER_OF_PELS:
1771         end_queue_message();
1772         temp = 0;
1773         for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0)
1774             for (slotno = 0; slotno < MAX_SLOT_COUNT; ++slotno)
1775                 if (system_config->lane[lane0]->pel[slotno] != NULL)
1776                     ++temp;
1777         lm_put_int(temp);
1778         break;
1779     case LM_NUMBER_OF_DABS:
1780         end_queue_message();
1781         temp = 0;
1782         for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i)
1783             if (dab_list[i] != NULL) {
1784                 temp += dab_list[i]->segment_count;
1785             }
1786         lm_put_int(temp);
1787         break;
1788     case LM_NUMBER_OF_PUBLIC_DEVICES:
1789         end_queue_message();
1790         temp = 0;
1791         for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i)
1792             if (dab_list[i] != NULL) {
1793                 if (dab_list[i]->used_as_private == FALSE) {
1794                     ++temp;
1795                 }
1796             }
1797     }
1798 }

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/function.c | DATE 5/23/89 | PAGE # 16/33 |
|--|---|-------------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 1799 | } | | | |
| 1800 | lm_put_int(temp); | | | |
| 1801 | break; | | | |
| 1802 | case LM_NUMBER_OF_PRIVATE_DEVICES: | | | |
| 1803 | end_queue_message(); | | | |
| 1804 | temp = 0; | | | |
| 1805 | for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i) | | | |
| 1806 | if (dab_list[i] != NULL) { | | | |
| 1807 | if (dab_list[i]->used_as_private == TRUE) { | | | |
| 1808 | ++temp; | | | |
| 1809 | } | | | |
| 1810 | } | | | |
| 1811 | lm_put_int(temp); | | | |
| 1812 | break; | | | |
| 1813 | case LM_NUMBER_OF_ACTIVE_INSTANCES: | | | |
| 1814 | end_queue_message(); | | | |
| 1815 | temp = 0; | | | |
| 1816 | for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i) | | | |
| 1817 | if (dab_list[i] != NULL) | | | |
| 1818 | temp += dab_list[i]->act_inst_count; | | | |
| 1819 | lm_put_int(temp); | | | |
| 1820 | break; | | | |
| 1821 | case LM_NUMBER_OF_ACTIVE_FAULTS: | | | |
| 1822 | end_queue_message(); | | | |
| 1823 | temp = 0; | | | |
| 1824 | for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i) | | | |
| 1825 | if (dab_list[i] != NULL) | | | |
| 1826 | temp += dab_list[i]->act_var_count; | | | |
| 1827 | lm_put_int(temp); | | | |
| 1828 | break; | | | |
| 1829 | case LM_SOFTWARE_REVISION_NUMBER: | | | |
| 1830 | end_queue_message(); | | | |
| 1831 | lm_put_int(SOFTWARE_REVISION_NUMBER); | | | |
| 1832 | break; | | | |
| 1833 | case LM_PASSWORD_CHECK: | | | |
| 1834 | if (Check_password(user) == TRUE) { | | | |
| 1835 | end_queue_message(); | | | |
| 1836 | lm_put_int(LM_TRUE); | | | |
| 1837 | } | | | |
| 1838 | else { | | | |
| 1839 | end_queue_message(); | | | |
| 1840 | lm_put_int(LM_FALSE); | | | |
| 1841 | } | | | |
| 1842 | break; | | | |
| 1843 | case LM_GET_VERSION_STRING_ADDR: | | | |
| 1844 | end_queue_message(); | | | |
| 1845 | temp = (u_long)lmsi_version; | | | |
| 1846 | lm_put_int(temp); | | | |
| 1847 | break; | | | |
| 1848 | default: | | | |
| 1849 | if (epoch_info(attrib, &value) == FAILURE) { | | | |
| 1850 | lm_queue_message(ERROR_MSG, "illegal attribute: %d", | | | |
| 1851 | attrib); | | | |
| 1852 | end_queue_message(); | | | |
| 1853 | } | | | |
| 1854 | else { | | | |
| 1855 | end_queue_message(); | | | |
| 1856 | lm_put_int(value); | | | |
| 1857 | } | | | |
| 1858 | break; | | | |
| 1859 | } | | | |
| 1860 | end_put(user->fd); | | | |
| 1861 | } | | | |
| 1862 | } | | | |
| 1863 | /* ARGSUSED */ | | | |
| 1864 | void process_lm_user_list_cmd(user) | | | |
| 1865 | USER_INFO *user; | | | |
| 1866 | { | | | |
| 1867 | u_short temp; | | | |
| 1868 | u_short j; | | | |
| 1869 | u_char i; | | | |
| 1870 | char c; | | | |
| 1871 | { | | | |
| 1872 | DPRINTF(("inside process_lm_user_list_cmd\n")); | | | |
| 1873 | { | | | |
| 1874 | reset_obuf(); | | | |
| 1875 | lm_put_int(LM_USER_LIST_MSG); | | | |
| 1876 | end_queue_message(); | | | |
| 1877 | { | | | |
| 1878 | temp = 0; | | | |
| 1879 | for (i = 0; i < MAX_USER_COUNT; ++i) | | | |
| 1880 | if (user_info_array[i]->active == TRUE) | | | |
| 1881 | ++temp; | | | |
| 1882 | } | | | |
| 1883 | lm_put_int(temp); | | | |
| 1884 | { | | | |
| 1885 | /* Put the user numbers to the Network buffer */ | | | |
| 1886 | for (i = 0; i < MAX_USER_COUNT; ++i) | | | |
| 1887 | if (user_info_array[i]->active == TRUE) | | | |
| 1888 | lm_put_int(i); | | | |
| 1889 | { | | | |
| 1890 | /* Put the hostnames to the Network buffer */ | | | |
| 1891 | for (i = 0; i < MAX_USER_COUNT; ++i) | | | |
| 1892 | if (user_info_array[i]->active == TRUE) { | | | |
| 1893 | for (j = 0; (c = user_info_array[i]->hostname[j]) != '\0'; ++j) | | | |
| 1894 | lm_put_char(c); | | | |
| 1895 | lm_put_char('\0'); | | | |
| 1896 | } | | | |
| 1897 | { | | | |
| 1898 | /* Put the usernames to the Network buffer */ | | | |
| 1899 | for (i = 0; i < MAX_USER_COUNT; ++i) | | | |
| 1900 | if (user_info_array[i]->active == TRUE) { | | | |
| 1901 | for (j = 0; (c = user_info_array[i]->username[j]) != '\0'; ++j) | | | |
| 1902 | lm_put_char(c); | | | |
| 1903 | lm_put_char('\0'); | | | |
| 1904 | } | | | |
| 1905 | { | | | |
| 1906 | end_put(user->fd); | | | |
| 1907 | } | | | |
| 1908 | { | | | |
| 1909 | /* ARGSUSED */ | | | |
| 1910 | void process_lm_user_cmd(user) | | | |
| 1911 | USER_INFO *user; | | | |
| 1912 | { | | | |
| 1913 | DEVICE_SPEC *definition; | | | |
| 1914 | INSTANCE_INFO *instance; | | | |
| 1915 | { | | | |
| 1916 | { | | | |
| 1917 | { | | | |
| 1918 | { | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89 PAGE #
TIME 6:14:38 pm 17/34

LINE # SOURCE TEXT

```

1919 USER INFO *userx;
1920 CONNECTION *conn;
1921 long userno;
1922 long attrib;
1923 u_long temp;
1924 u_char i;
1925
1926 DPRINTF(("inside process_lm_user_cmd\n"));
1927
1928 reset_out();
1929 lm_put_int(IMO_USER_AMS);
1930
1931 userno = lm_get_int();
1932 attrib = lm_get_int();
1933
1934 if (userno >= MAX_USER_COUNT) {
1935     lm_queue_message(ERROR_MSG, "invalid user number: id specified",
1936                     userno);
1937     end_queue_message();
1938     end_put(user->id);
1939     return;
1940 }
1941
1942 if (user_info_array[userno]->active == FALSE) {
1943     lm_queue_message(ERROR_MSG, "user number: id does not exist",
1944                     userno);
1945     end_queue_message();
1946     end_put(user->id);
1947     return;
1948 }
1949
1950 userx = user_info_array[userno];
1951
1952 switch (attrib) {
1953 case LM_NUMBER_OF_PATTERNS_ALLOCATED:
1954     end_queue_message();
1955     temp = 0;
1956     for (i = 0; i < userx->inst_table_size; ++i) {
1957         instance = userx->instance[i];
1958         if (BOGUS_INSTANCE(userx, instance))
1959             continue;
1960
1961         if (instance->is_fault == FALSE)
1962             temp += ((instance->pattern_count + PTRN_PER_BLOCK) /
1963                     PTRN_PER_BLOCK) * PTRN_PER_BLOCK;
1964         else
1965             temp += ((instance->pattern_count -
1966                     instance->common_pattern_count + PTRN_PER_BLOCK) /
1967                     PTRN_PER_BLOCK) * PTRN_PER_BLOCK;
1968     }
1969     lm_put_int(temp);
1970     break;
1971 case LM_NUMBER_OF_PATTERNS:
1972     end_queue_message();
1973     temp = 0;
1974     for (i = 0; i < userx->inst_table_size; ++i) {
1975         instance = userx->instance[i];
1976         if (BOGUS_INSTANCE(userx, instance))
1977             continue;
1978
1979         if (instance->is_fault == FALSE)
1980             temp += instance->pattern_count;
1981         else
1982             temp += instance->pattern_count -
1983                     instance->common_pattern_count;
1984     }
1985     lm_put_int(temp);
1986     break;
1987 case LM_NUMBER_OF_PUBLIC_DEVICES:
1988     end_queue_message();
1989     temp = 0;
1990     for (i = 0; i < userx->def_table_size; ++i) {
1991         definition = userx->definition[i];
1992         if (BOGUS_DEFINITION(userx, definition))
1993             continue;
1994
1995         if (definition->device_type == PUBLIC)
1996             ++temp;
1997     }
1998     lm_put_int(temp);
1999     break;
2000 case LM_NUMBER_OF_PRIVATE_DEVICES:
2001     end_queue_message();
2002     temp = 0;
2003     for (i = 0; i < userx->def_table_size; ++i) {
2004         definition = userx->definition[i];
2005
2006         if (BOGUS_DEFINITION(userx, definition))
2007             continue;
2008
2009         if (definition->device_type == PRIVATE)
2010             ++temp;
2011     }
2012     lm_put_int(temp);
2013     break;
2014 case LM_NUMBER_OF_ACTIVE_INSTANCES:
2015     end_queue_message();
2016     temp = 0;
2017     for (i = 0; i < userx->inst_table_size; ++i) {
2018         instance = userx->instance[i];
2019         if (BOGUS_INSTANCE(userx, instance))
2020             continue;
2021
2022         if (instance->is_fault == FALSE)
2023             ++temp;
2024     }
2025     lm_put_int(temp);
2026     break;
2027 case LM_NUMBER_OF_ACTIVE_FAULTS:
2028     end_queue_message();
2029     temp = 0;
2030     for (i = 0; i < userx->inst_table_size; ++i) {
2031         instance = userx->instance[i];
2032
2033         if (BOGUS_INSTANCE(userx, instance))
2034             continue;
2035
2036         if (instance->is_fault == TRUE)
2037             ++temp;
2038     }

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/function.c | DATE 5/23/89 | PAGE # 18/35 |
|--|--|---|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 2039 | | lm_put_int(temp); | | |
| 2040 | | break; | | |
| 2041 | | case LM_SECONDS_SINCE_LAST_RESPONSE: | | |
| 2042 | | end_queue_message(); | | |
| 2043 | | conn = table_of_conns(userno); | | |
| 2044 | | lm_put_int((lm_tick - conn->time_at_last_response) / TICKS_PER_SECOND); | | |
| 2045 | | break; | | |
| 2046 | | default: | | |
| 2047 | | lm_queue_message(ERROR_MSG, "illegal attribute: %d", | | |
| 2048 | | attrib); | | |
| 2049 | | end_queue_message(); | | |
| 2050 | | break; | | |
| 2051 | | } | | |
| 2052 | | end_put(user->fd); | | |
| 2053 | | } | | |
| 2054 | | | | |
| 2055 | | /* ARGSUSED */ | | |
| 2056 | | void process_inq_lane_cmd(user) | | |
| 2057 | | USER_INFO *user; | | |
| 2058 | | { | | |
| 2059 | | long lanesum; | | |
| 2060 | | u_char panno; | | |
| 2061 | | u_char slotno; | | |
| 2062 | | u_long temp; | | |
| 2063 | | u_long attribute; | | |
| 2064 | | | | |
| 2065 | | DPRINTF(("inside process_inq_lane_cmd\n")); | | |
| 2066 | | reset_obuf(); | | |
| 2067 | | lm_put_int(INQ_LANE_ANS); | | |
| 2068 | | | | |
| 2069 | | lanesum = lm_get_int(); | | |
| 2070 | | if ((lanesum < 0) | | |
| 2071 | | (lanesum >= system_config->phy_lane_count)) { | | |
| 2072 | | lm_queue_message(ERROR_MSG, "illegal lane: %d", | | |
| 2073 | | lanesum); | | |
| 2074 | | end_queue_message(); | | |
| 2075 | | break; | | |
| 2076 | | | | |
| 2077 | | end_put(user->fd); | | |
| 2078 | | return; | | |
| 2079 | | } | | |
| 2080 | | | | |
| 2081 | | attribute = lm_get_int(); | | |
| 2082 | | | | |
| 2083 | | if ((attribute != LM_IS_LANE_USABLE) && | | |
| 2084 | | (system_config->lane[lanesum]->pac_present == FALSE)) { | | |
| 2085 | | lm_queue_message(ERROR_MSG, "lane: %d is not usable", | | |
| 2086 | | 'A' + lanesum); | | |
| 2087 | | end_queue_message(); | | |
| 2088 | | break; | | |
| 2089 | | end_put(user->fd); | | |
| 2090 | | return; | | |
| 2091 | | } | | |
| 2092 | | | | |
| 2093 | | switch (attribute) { | | |
| 2094 | | case LM_IS_LANE_USABLE: | | |
| 2095 | | end_queue_message(); | | |
| 2096 | | | | |
| 2097 | | if (system_config->lane[lanesum]->pac_present == TRUE) | | |
| 2098 | | lm_put_int(LM_TRUE); | | |
| 2099 | | else | | |
| 2100 | | lm_put_int(LM_FALSE); | | |
| 2101 | | break; | | |
| 2102 | | case LM_TOTAL_PATTERNS: | | |
| 2103 | | end_queue_message(); | | |
| 2104 | | | | |
| 2105 | | temp = 0; | | |
| 2106 | | for (panno = 0; panno < MAX_PAN_COUNT; ++panno) | | |
| 2107 | | if (system_config->lane[lanesum]->pan[panno] != NULL) | | |
| 2108 | | temp += system_config->lane[lanesum]->pan[panno]->mem_size; | | |
| 2109 | | lm_put_int(temp); | | |
| 2110 | | break; | | |
| 2111 | | case LM_AVAILABLE_PATTERNS: | | |
| 2112 | | end_queue_message(); | | |
| 2113 | | temp = count_avail_patterns((u_char)lanesum) * PTEN_PER_BLOCK; | | |
| 2114 | | lm_put_int(temp); | | |
| 2115 | | break; | | |
| 2116 | | case LM_NUMBER_OF_PANS: | | |
| 2117 | | end_queue_message(); | | |
| 2118 | | | | |
| 2119 | | temp = 0; | | |
| 2120 | | for (panno = 0; panno < MAX_PAN_COUNT; ++panno) | | |
| 2121 | | if (system_config->lane[lanesum]->pan[panno] != NULL) | | |
| 2122 | | ++temp; | | |
| 2123 | | lm_put_int(temp); | | |
| 2124 | | break; | | |
| 2125 | | case LM_NUMBER_OF_PELS: | | |
| 2126 | | end_queue_message(); | | |
| 2127 | | | | |
| 2128 | | temp = 0; | | |
| 2129 | | for (slotno = 0; slotno < MAX_SLOT_COUNT; ++slotno) | | |
| 2130 | | if (system_config->lane[lanesum]->pel[slotno] != NULL) | | |
| 2131 | | ++temp; | | |
| 2132 | | lm_put_int(temp); | | |
| 2133 | | break; | | |
| 2134 | | case LM_NUMBER_OF_POPULATED_PELS: | | |
| 2135 | | end_queue_message(); | | |
| 2136 | | | | |
| 2137 | | lm_put_int(calculate_pel_count((u_char)lanesum)); | | |
| 2138 | | break; | | |
| 2139 | | default: | | |
| 2140 | | lm_queue_message(ERROR_MSG, "illegal attribute: %d", | | |
| 2141 | | attribute); | | |
| 2142 | | end_queue_message(); | | |
| 2143 | | break; | | |
| 2144 | | } | | |
| 2145 | | end_put(user->fd); | | |
| 2146 | | } | | |
| 2147 | | | | |
| 2148 | | /* ARGSUSED */ | | |
| 2149 | | void process_inq_pan_cmd(user) | | |
| 2150 | | USER_INFO *user; | | |
| 2151 | | { | | |
| 2152 | | u_long panno; | | |
| 2153 | | u_long lanesum; | | |
| 2154 | | u_char attrib; | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/function.c

DATE 5/23/89

PAGE #

TIME 6:14:38 pm

19/36

```

LINE # SOURCE TEXT
2159 DPRINTF(("inside process_inq_pam_cmd\n"));
2160
2161 reset_obuf();
2162 lm_put_int(INQ_PAM_ANS);
2163 laneso = lm_get_int();
2164 panno = lm_get_int();
2165
2166 if ((laneso < 0) || (laneso >= MAX_LANE_COUNT)) {
2167     lm_queue_message(ERROR_MSG, "illegal lane: %d",
2168                     laneso);
2169     end_queue_message();
2170     end_put(user->id);
2171     return;
2172 }
2173
2174 if ((panno < 0) || (panno >= MAX_PAM_COUNT)) {
2175     lm_queue_message(ERROR_MSG, "illegal Fast Pattern Memory: %d",
2176                     panno);
2177     end_queue_message();
2178     end_put(user->id);
2179     return;
2180 }
2181
2182 if (system_config->lane[laneso]->pac_present == FALSE) {
2183     lm_queue_message(ERROR_MSG, "lane: %c does not exist",
2184                     'A' + laneso);
2185     end_queue_message();
2186     end_put(user->id);
2187     return;
2188 }
2189
2190 if (system_config->lane[laneso]->pam[panno] == NULL) {
2191     lm_queue_message(ERROR_MSG, "Fast Pattern Memory: %d in lane: %c does not exist",
2192                     panno, 'A' + laneso);
2193     end_queue_message();
2194     end_put(user->id);
2195     return;
2196 }
2197
2198 switch (attrib = lm_get_int()) {
2199 case LM_PAM_MEMORY_SIZE:
2200     end_queue_message();
2201     lm_put_int(system_config->lane[laneso]->pam[panno]->mem_size);
2202     break;
2203 default:
2204     lm_queue_message(ERROR_MSG, "illegal attribute: %d",
2205                     attrib);
2206     end_queue_message();
2207     break;
2208 }
2209
2210 end_put(user->id);
2211 }
2212
2213 /* ARGSUSED */
2214 void process_inq_pel_cmd(user)
2215 USER_INFO *user;
2216 {
2217     DAB_INFO *dab_ptr;
2218     long laneso;
2219     long slotno;
2220     u_char dabno;
2221     u_char unitno;
2222     u_char attrib;
2223
2224     DPRINTF(("inside process_inq_pel_cmd\n"));
2225
2226     reset_obuf();
2227     lm_put_int(INQ_PEL_ANS);
2228
2229     laneso = lm_get_int();
2230     slotno = lm_get_int();
2231     attrib = lm_get_int();
2232
2233     if ((laneso < 0) || (laneso >= MAX_LANE_COUNT)) {
2234         lm_queue_message(ERROR_MSG, "illegal lane: %d",
2235                         laneso);
2236         end_queue_message();
2237         end_put(user->id);
2238         return;
2239     }
2240
2241     if ((slotno < 0) || (slotno >= MAX_SLOT_COUNT)) {
2242         lm_queue_message(ERROR_MSG, "illegal slot: %d",
2243                         slotno);
2244         end_queue_message();
2245         end_put(user->id);
2246         return;
2247     }
2248
2249     if (system_config->lane[laneso]->pel[slotno] == NULL) {
2250         if (attrib == LM_DOES_PEL_EXIST) {
2251             end_queue_message();
2252             lm_put_int(LM_FALSE);
2253         }
2254         else {
2255             lm_queue_message(ERROR_MSG, "Pin Electronics Module in lane: %c slot: %d does not exist",
2256                             'A' + laneso, slotno);
2257             end_queue_message();
2258         }
2259         end_put(user->id);
2260         return;
2261     }
2262
2263     switch (attrib) {
2264     case LM_DOES_PEL_EXIST:
2265         end_queue_message();
2266         lm_put_int(LM_TRUE);
2267         break;
2268     case LM_IS_DAB_PRESENT:
2269         for (dabno = 0; dabno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++dabno) {
2270             dab_ptr = dab_list[dabno];
2271             if (dab_ptr == NULL)
2272                 continue;
2273
2274             for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
2275                 if ((dab_ptr->unit_location[unitno].lane_no == laneso) &&

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89
TIME 6:14:38 pm

PAGE #
20/37

```

LINE #          SOURCE TEXT
2279          (dab_ptr->unit_location[unitno].slot_no == slotno)) {
2280          end_queue_message();
2281          lm_put_int(LM_TRUE);
2282          end_put(user->id);
2283          return;
2284          }
2285          }
2286          }
2287          }
2288          end_queue_message();
2289          lm_put_int(LM_FALSE);
2290          break;
2291          default:
2292          lm_queue_message(ERROR_MSG, "illegal attribute: %d",
2293          attrib);
2294          end_queue_message();
2295          break;
2296          }
2297          }
2298          end_put(user->id);
2299          }
2300          }
2301          /* ARGSUSED */
2302          void process_inq_device_list_cmd(user)
2303          USER_INFO *user;
2304          {
2305          {
2306          u_long   dab_count_mark;
2307          u_char   dab_count;
2308          u_char   dabno;
2309          u_char   i;
2310          char     c;
2311          }
2312          DPRINTF(("inside process_inq_device_list_cmd\n"));
2313          reset_obuf();
2314          lm_put_int(INQ_DEVICE_LIST_ANS);
2315          end_queue_message();
2316          }
2317          dab_count_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
2318          lm_put_int(0); /* dummy dab_count */
2319          }
2320          }
2321          dab_count = 0;
2322          for (dabno = 0; dabno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++dabno) {
2323          if (dab_list[dabno] != NULL) {
2324          --dab_count;
2325          lm_put_int((u_long)dabno);
2326          }
2327          }
2328          }
2329          for (dabno = 0; dabno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++dabno) {
2330          if (dab_list[dabno] != NULL) {
2331          for (i = 0; (c = dab_list[dabno]->part_name[i]) != '\0'; ++i)
2332          lm_put_char(c);
2333          lm_put_char('\0');
2334          }
2335          }
2336          }
2337          LM_PUT_LONG_AT_MARK(dab_count_mark, lm_global_conn_ptr, (u_long)dab_count);
2338          }
2339          end_put(user->id);
2340          return;
2341          }
2342          }
2343          /* ARGSUSED */
2344          void process_inq_device_name_cmd(user)
2345          USER_INFO *user;
2346          {
2347          DAB_INFO *dab_ptr;
2348          u_long   device_no;
2349          u_char   i;
2350          char     c;
2351          }
2352          DPRINTF(("inside process_inq_device_name_cmd\n"));
2353          reset_obuf();
2354          lm_put_int(INQ_DEVICE_NAME_ANS);
2355          }
2356          device_no = lm_get_int();
2357          if (device_no >= (MAX_LANE_COUNT * MAX_SLOT_COUNT)) {
2358          lm_queue_message(ERROR_MSG, "invalid device number: %d specified",
2359          device_no);
2360          end_queue_message();
2361          end_put(user->id);
2362          return;
2363          }
2364          }
2365          }
2366          dab_ptr = dab_list[device_no];
2367          if (dab_ptr == NULL) {
2368          lm_queue_message(ERROR_MSG, "invalid device number: %d specified",
2369          device_no);
2370          end_queue_message();
2371          end_put(user->id);
2372          return;
2373          }
2374          }
2375          end_queue_message();
2376          }
2377          for (i = 0; c = dab_ptr->part_name[i]; ++i)
2378          lm_put_char(c);
2379          lm_put_char('\0');
2380          }
2381          end_put(user->id);
2382          }
2383          }
2384          /* ARGSUSED */
2385          void process_inq_device_cmd(user)
2386          USER_INFO *user;
2387          {
2388          USER_INFO *userx;
2389          DAB_INFO *dab_ptr;
2390          DAB_INFO *temp;
2391          INSTANCE_INFO *instance;
2392          EXTRA_DEVICE_SPEC *extra_def_ptr;
2393          u_long   device_no;
2394          u_long   total_patterns;
2395          u_short   inst_id;
2396          u_char   attrib;
2397          u_char   userno;
2398          }

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:38 pm | 21/38 |

| LINE # | SOURCE TEXT |
|--------|--|
| 2399 | DPRINTF(("inside process_lm_device_cmd\n")); |
| 2400 | |
| 2401 | reset_obuf(); |
| 2402 | lm_put_int(lmq_DEVICE_ANS); |
| 2403 | |
| 2404 | device_no = lm_get_int(); |
| 2405 | if (device_no >= (MAX_LANE_COUNT * MAX_SLOT_COUNT)) { |
| 2406 | lm_queue_message(ERROR_MSG, "invalid device number: %d specified", |
| 2407 | device_no); |
| 2408 | end_queue_message(); |
| 2409 | end_put(user->fd); |
| 2410 | return; |
| 2411 | } |
| 2412 | |
| 2413 | temp = dab_list[device_no]; |
| 2414 | if (temp == NULL) { |
| 2415 | lm_queue_message(ERROR_MSG, "invalid device number: %d specified", |
| 2416 | device_no); |
| 2417 | end_queue_message(); |
| 2418 | end_put(user->fd); |
| 2419 | return; |
| 2420 | } |
| 2421 | |
| 2422 | switch (attrib = lm_get_int()) { |
| 2423 | case LM_DEVICE_STATUS: |
| 2424 | end_queue_message(); |
| 2425 | if (temp->used_as_private == TRUE) |
| 2426 | lm_put_int(LM_PRIVATE); |
| 2427 | else |
| 2428 | lm_put_int(LM_PUBLIC); |
| 2429 | break; |
| 2430 | case LM_NUMBER_OF_DABS: |
| 2431 | end_queue_message(); |
| 2432 | lm_put_int(temp->segment_count); |
| 2433 | break; |
| 2434 | case LM_NUMBER_OF_ACTIVE_INSTANCES: |
| 2435 | end_queue_message(); |
| 2436 | lm_put_int(temp->act_inst_count); |
| 2437 | break; |
| 2438 | case LM_NUMBER_OF_ACTIVE_FAULTS: |
| 2439 | end_queue_message(); |
| 2440 | lm_put_int(temp->act_var_count); |
| 2441 | break; |
| 2442 | case LM_NUMBER_OF_PATTERNS: |
| 2443 | total_patterns = 0; |
| 2444 | for (userno = 0; userno < MAX_USER_COUNT; ++userno) { |
| 2445 | userx = user_info_array[userno]; |
| 2446 | |
| 2447 | if (userx->active == FALSE) |
| 2448 | continue; |
| 2449 | |
| 2450 | for (inst_id = 0; inst_id < userx->inst_table_size; ++inst_id) { |
| 2451 | instance = userx->instance[inst_id]; |
| 2452 | |
| 2453 | if (BOGUS_INSTANCE(userx, instance)) |
| 2454 | continue; |
| 2455 | |
| 2456 | extra_def_ptr = (EXTRA_DEVICE_SPEC *) |
| 2457 | instance->definition->extra_data; |
| 2458 | |
| 2459 | if (extra_def_ptr->dab_ok == FALSE) |
| 2460 | continue; |
| 2461 | |
| 2462 | dab_ptr = dab_list[instance->dab_info_index]; |
| 2463 | |
| 2464 | /* If this instance is using the device we are looking for then |
| 2465 | * count the pattern usage. |
| 2466 | */ |
| 2467 | if (dab_ptr == temp) { |
| 2468 | if (instance->is_fault == FALSE) { |
| 2469 | total_patterns += instance->pattern_count; |
| 2470 | } |
| 2471 | else { |
| 2472 | total_patterns += instance->pattern_count - |
| 2473 | instance->common_pattern_count; |
| 2474 | } |
| 2475 | } |
| 2476 | |
| 2477 | end_queue_message(); |
| 2478 | lm_put_int(total_patterns); |
| 2479 | break; |
| 2480 | default: |
| 2481 | lm_queue_message(ERROR_MSG, "illegal attribute: %d", |
| 2482 | attrib); |
| 2483 | end_queue_message(); |
| 2484 | break; |
| 2485 | |
| 2486 | end_put(user->fd); |
| 2487 | |
| 2488 | |
| 2489 | |
| 2490 | /* ARGSUSED */ |
| 2491 | void process_lm_dab_cmd(user) |
| 2492 | USER_INFO *user; |
| 2493 | { |
| 2494 | |
| 2495 | DAB_INFO *temp; |
| 2496 | SEGMENT_EL *seg_ptr; |
| 2497 | u_long device_no; |
| 2498 | long dab_number; |
| 2499 | char str[16]; |
| 2500 | u_char i; |
| 2501 | u_char attrib; |
| 2502 | char c; |
| 2503 | |
| 2504 | DPRINTF(("inside process_lm_dab_cmd\n")); |
| 2505 | |
| 2506 | reset_obuf(); |
| 2507 | lm_put_int(lmq_DAB_ANS); |
| 2508 | |
| 2509 | device_no = lm_get_int(); |
| 2510 | if (device_no >= (MAX_LANE_COUNT * MAX_SLOT_COUNT)) { |
| 2511 | lm_queue_message(ERROR_MSG, "invalid device number: %d specified", |
| 2512 | device_no); |
| 2513 | end_queue_message(); |
| 2514 | end_put(user->fd); |
| 2515 | return; |
| 2516 | } |
| 2517 | |
| 2518 | temp = dab_list[device_no]; |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/function.c | DATE 5/23/89 | PAGE # 22/39 |
|--|--|---|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 2519 | | if (temp == NULL) { | | |
| 2520 | | lm_queue_message(ERROR_MSG, "invalid device number: %d specified", | | |
| 2521 | | device_no); | | |
| 2522 | | end_queue_message(); | | |
| 2523 | | end_put(user->fd); | | |
| 2524 | | return; | | |
| 2525 | | } | | |
| 2526 | | dab_number = lm_get_int(); | | |
| 2527 | | if (dab_number >= temp->segment_count) { | | |
| 2528 | | lm_queue_message(ERROR_MSG, "illegal Device Adapter number: %d", | | |
| 2529 | | dab_number); | | |
| 2530 | | end_queue_message(); | | |
| 2531 | | end_put(user->fd); | | |
| 2532 | | return; | | |
| 2533 | | } | | |
| 2534 | | if (temp->segment[0] != NULL) | | |
| 2535 | | seg_ptr = temp->segment[0]; | | |
| 2536 | | else { | | |
| 2537 | | seg_ptr = temp->segment(dab_number + 1); | | |
| 2538 | | if (seg_ptr == NULL) { | | |
| 2539 | | lm_queue_message(ERROR_MSG, "internal error: segment D/S not found"); | | |
| 2540 | | end_queue_message(); | | |
| 2541 | | end_put(user->fd); | | |
| 2542 | | return; | | |
| 2543 | | } | | |
| 2544 | | } | | |
| 2545 | | switch (attrib = lm_get_int()) { | | |
| 2546 | | case LM_DAB_TYPE: | | |
| 2547 | | end_queue_message(); | | |
| 2548 | | for (i = 0; c = seg_ptr->dab_type[i]; ++i) | | |
| 2549 | | lm_put_char(c); | | |
| 2550 | | lm_put_char('\0'); | | |
| 2551 | | break; | | |
| 2552 | | case LM_DAB_REVISION: | | |
| 2553 | | end_queue_message(); | | |
| 2554 | | for (i = 0; c = seg_ptr->revision[i]; ++i) | | |
| 2555 | | lm_put_char(c); | | |
| 2556 | | lm_put_char('\0'); | | |
| 2557 | | break; | | |
| 2558 | | case LM_DAB_MANUFACTURER: | | |
| 2559 | | end_queue_message(); | | |
| 2560 | | for (i = 0; c = seg_ptr->man_id[i]; ++i) | | |
| 2561 | | lm_put_char(c); | | |
| 2562 | | lm_put_char('\0'); | | |
| 2563 | | break; | | |
| 2564 | | case LM_DAB_MODEL: | | |
| 2565 | | end_queue_message(); | | |
| 2566 | | for (i = 0; c = seg_ptr->model_id[i]; ++i) | | |
| 2567 | | lm_put_char(c); | | |
| 2568 | | lm_put_char('\0'); | | |
| 2569 | | break; | | |
| 2570 | | case LM_DAB_MODEL_REVISION: | | |
| 2571 | | end_queue_message(); | | |
| 2572 | | for (i = 0; c = seg_ptr->model_revision[i]; ++i) | | |
| 2573 | | lm_put_char(c); | | |
| 2574 | | lm_put_char('\0'); | | |
| 2575 | | break; | | |
| 2576 | | case LM_DAB_INSERTIONS: | | |
| 2577 | | end_queue_message(); | | |
| 2578 | | (void)sprintf(str, "%d", seg_ptr->insertion_count); | | |
| 2579 | | for (i = 0; c = str[i]; ++i) | | |
| 2580 | | lm_put_char(c); | | |
| 2581 | | lm_put_char('\0'); | | |
| 2582 | | break; | | |
| 2583 | | default: | | |
| 2584 | | lm_queue_message(ERROR_MSG, "illegal attribute: %d", | | |
| 2585 | | attrib); | | |
| 2586 | | end_queue_message(); | | |
| 2587 | | break; | | |
| 2588 | | } | | |
| 2589 | | end_put(user->fd); | | |
| 2590 | | } | | |
| 2591 | | /* ABCSTUD */ | | |
| 2592 | | void process_lm_dab_loc_cmd(user) | | |
| 2593 | | USER_INFO *user; | | |
| 2594 | | { | | |
| 2595 | | DAB_INFO *temp; | | |
| 2596 | | u_long device_no; | | |
| 2597 | | u_long dab_no; | | |
| 2598 | | DPRINTF(("inside process_lm_dab_loc_cmd\n")); | | |
| 2599 | | resetobuf(); | | |
| 2600 | | lm_put_int(ING_DAB_LOC_AMS); | | |
| 2601 | | device_no = lm_get_int(); | | |
| 2602 | | dab_no = lm_get_int(); | | |
| 2603 | | if (device_no >= (MAX_LANE_COUNT * MAX_SLOT_COUNT)) { | | |
| 2604 | | lm_queue_message(ERROR_MSG, "invalid device number: %d specified", | | |
| 2605 | | device_no); | | |
| 2606 | | end_queue_message(); | | |
| 2607 | | end_put(user->fd); | | |
| 2608 | | return; | | |
| 2609 | | } | | |
| 2610 | | temp = dab_list(device_no); | | |
| 2611 | | if (temp == NULL) { | | |
| 2612 | | lm_queue_message(ERROR_MSG, "invalid device number: %d specified", | | |
| 2613 | | device_no); | | |
| 2614 | | end_queue_message(); | | |
| 2615 | | end_put(user->fd); | | |
| 2616 | | return; | | |
| 2617 | | } | | |
| 2618 | | } | | |
| 2619 | | } | | |
| 2620 | | } | | |
| 2621 | | } | | |
| 2622 | | } | | |
| 2623 | | } | | |
| 2624 | | } | | |
| 2625 | | } | | |
| 2626 | | } | | |
| 2627 | | } | | |
| 2628 | | } | | |
| 2629 | | } | | |
| 2630 | | } | | |
| 2631 | | } | | |
| 2632 | | } | | |
| 2633 | | } | | |
| 2634 | | } | | |
| 2635 | | } | | |
| 2636 | | } | | |
| 2637 | | } | | |
| 2638 | | } | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89
TIME 6:14:38 pm

PAGE #
23/40

```

LINE #          SOURCE TEXT
2639      end_put(user->id);
2640      return;
2641  }
2642
2643      if (temp->segment[0] != NULL) {
2644          /* Single segment case */
2645          if (dab_no != 0) {
2646              lm_queue_message(ERROR_MSG, "invalid Device Adapter number: %d specified",
2647                              dab_no);
2648              end_queue_message();
2649              end_put(user->id);
2650              return;
2651          }
2652      }
2653      else {
2654          /* Multi segment Device */
2655          if (dab_no != temp->segment_count) {
2656              lm_queue_message(ERROR_MSG, "invalid Device Adapter number: %d specified",
2657                              dab_no);
2658              end_queue_message();
2659              end_put(user->id);
2660              return;
2661          }
2662      }
2663      end_queue_message();
2664      dab_loc(temp, dab_no);
2665      end_put(user->id);
2666  }
2667
2668  void process_inq_instance_cmd(user)
2669  USER_INFO *user;
2670  {
2671      INSTANCE_INFO *instance;
2672      u_long      freq;
2673      u_short     inst_id;
2674      u_short     attrib;
2675
2676      DPRINTF(("inside process_inq_instance_cmd\n"));
2677
2678      reset_obuf();
2679      lm_put_inst(INQ_INSTANCE_ANS);
2680
2681      inst_id = lm_get_short();
2682      attrib = lm_get_inst();
2683
2684      /* verify inst_id */
2685      if (inst_id >= user->inst_table_size) {
2686          lm_queue_message(ERROR_MSG, "invalid instance id: %d specified",
2687                          inst_id);
2688          end_queue_message();
2689          end_put(user->id);
2690          return;
2691      }
2692
2693      instance = user->instance[inst_id];
2694      if (BOGUS_INSTANCE(user, instance)) {
2695          lm_queue_message(ERROR_MSG, "invalid instance id: %d specified",
2696                          inst_id);
2697          end_queue_message();
2698          end_put(user->id);
2699          return;
2700      }
2701
2702      if (instance->is_fault) {
2703          lm_queue_message(ERROR_MSG, "instance id: %d is a fault",
2704                          inst_id);
2705          end_queue_message();
2706          end_put(user->id);
2707          return;
2708      }
2709
2710      switch (attrib) {
2711      case LM_NUMBER_OF_PATTERNS:
2712          end_queue_message();
2713          lm_put_inst(instance->pattern_count);
2714          break;
2715      case LM_PATTERN_FREQUENCY:
2716          end_queue_message();
2717          freq = (double)1000000000000.0 / (double)((EXTRA_DEVICE_SPEC *)
2718              (instance->definition->extra_data))->actual_phy_clock_period + 0.5;
2719          lm_put_inst(freq);
2720          break;
2721      default:
2722          lm_queue_message(ERROR_MSG, "illegal attribute: %d",
2723                          attrib);
2724          end_queue_message();
2725          break;
2726      }
2727
2728      end_put(user->id);
2729  }
2730
2731  void process_inq_fault_cmd(user)
2732  USER_INFO *user;
2733  {
2734      INSTANCE_INFO *fault;
2735      u_long      freq;
2736      u_short     fault_id;
2737      u_short     attrib;
2738
2739      DPRINTF(("inside process_inq_fault_cmd\n"));
2740
2741      reset_obuf();
2742      lm_put_inst(INQ_FAULT_ANS);
2743
2744      fault_id = lm_get_short();
2745      attrib = lm_get_inst();
2746
2747      /* verify fault_id */
2748      if (fault_id >= user->inst_table_size) {
2749          lm_queue_message(ERROR_MSG, "invalid fault id: %d specified",
2750                          fault_id);
2751          end_queue_message();
2752          end_put(user->id);
2753          return;
2754      }
2755
2756      fault = user->fault[fault_id];
2757      if (BOGUS_FAULT(user, fault)) {
2758          lm_queue_message(ERROR_MSG, "invalid fault id: %d specified",
2759                          fault_id);
2760          end_queue_message();
2761          end_put(user->id);
2762          return;
2763      }
2764
2765      if (fault->is_fault) {
2766          lm_queue_message(ERROR_MSG, "fault id: %d is a fault",
2767                          fault_id);
2768          end_queue_message();
2769          end_put(user->id);
2770          return;
2771      }
2772
2773      switch (attrib) {
2774      case LM_NUMBER_OF_PATTERNS:
2775          end_queue_message();
2776          lm_put_inst(fault->pattern_count);
2777          break;
2778      case LM_PATTERN_FREQUENCY:
2779          end_queue_message();
2780          freq = (double)1000000000000.0 / (double)((EXTRA_DEVICE_SPEC *)
2781              (fault->definition->extra_data))->actual_phy_clock_period + 0.5;
2782          lm_put_inst(freq);
2783          break;
2784      default:
2785          lm_queue_message(ERROR_MSG, "illegal attribute: %d",
2786                          attrib);
2787          end_queue_message();
2788          break;
2789      }
2790
2791      end_put(user->id);
2792  }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89
TIME 6:14:38 pm

PAGE #
24/41

```

LINE # SOURCE TEXT
2759 fault = user->instance[fault_id];
2760 if (BOGUS_INSTANCE(user, fault)) {
2761     lm_queue_message(ERROR_MSG, "invalid fault id: %d specified",
2762         fault_id);
2763     end_queue_message();
2764     return;
2765 }
2766
2767 if (! fault->is_fault) {
2768     lm_queue_message(ERROR_MSG, "fault id: %d is an instance",
2769         fault_id);
2770     end_queue_message();
2771     return;
2772 }
2773
2774 switch (attrib) {
2775     case LM_NUMBER_OF_PATTERNS:
2776         end_queue_message();
2777         lm_put_int(fault->patterns_count);
2778         break;
2779     case LM_PATTERN_FREQUENCY:
2780         end_queue_message();
2781         freq = (double)1000000000000.0 / ((double)((EXTRA_DEVICE_SPEC *)
2782             (fault->definition->extra_data))->actual_phy_clock_period + 0.5);
2783         lm_put_int(freq);
2784         break;
2785     default:
2786         lm_queue_message(ERROR_MSG, "illegal attribute: %d",
2787             attrib);
2788         end_queue_message();
2789         break;
2790 }
2791
2792 end_put(user->fd);
2793
2794 void process_lm_avail_ptrn_cmd(user)
2795 USER_INFO *user;
2796 {
2797     DEVICE_SPEC *def_ptr;
2798     DAB_INFO *dab_ptr;
2799     long temp;
2800     long min_lane_ptrn_count;
2801     u_short def_id;
2802     char dab_info_index;
2803     u_char lanes;
2804
2805     /* Calculate the maximum number of patterns this definition can have.
2806      * This is used to RESTORE to first check if we can allocate the patterns
2807      * before actually sending the patterns.
2808      */
2809     DPRINTF(("inside process_lm_avail_ptrn_cmd\n"));
2810
2811     reset_chan();
2812     lm_put_int(LM_AVAIL_PTRN_ANS);
2813
2814     def_id = lm_get_short();
2815
2816     /* verify def_id */
2817     if (def_id >= user->def_table_size) {
2818         lm_queue_message(ERROR_MSG, "invalid definition id: %d specified",
2819             def_id);
2820         end_queue_message();
2821         end_put(user->fd);
2822         return;
2823     }
2824
2825     def_ptr = user->definition[def_id];
2826     if (BOGUS_DEFINITION(user, def_ptr)) {
2827         lm_queue_message(ERROR_MSG, "invalid definition id: %d specified",
2828             def_id);
2829         end_queue_message();
2830         end_put(user->fd);
2831         return;
2832     }
2833
2834     if ((short)(dab_info_index = find_dab(dab_list, def_ptr->device_name,
2835         (u_char)def_ptr->device_type)) == -1) {
2836         lm_queue_message(ERROR_MSG, "device name: %s not found or being used as PRIVATE",
2837             def_ptr->device_name);
2838         end_queue_message();
2839         end_put(user->fd);
2840         return;
2841     }
2842
2843     dab_ptr = dab_list[dab_info_index];
2844
2845     min_lane_ptrn_count = MAXINT;
2846
2847     for (lanes = 0; lanes < MAX_LANE_COUNT; ++lanes) {
2848         if (dab_ptr->lane_used[lanes] == FALSE)
2849             continue;
2850
2851         temp = count_avail_patterns((u_char)lanes) * PTRN_PER_BLOCK;
2852
2853         if (temp < min_lane_ptrn_count)
2854             min_lane_ptrn_count = temp;
2855     }
2856
2857     end_queue_message();
2858     lm_put_int(min_lane_ptrn_count);
2859     end_put(user->fd);
2860
2861 void process_tmmeasurement_cmd(user)
2862 USER_INFO *user;
2863 {
2864     DEVICE_SPEC *def_ptr;
2865     INSTANCE_INFO *instance;
2866     u_short inst_id;
2867     u_char on_or_off;
2868
2869     DPRINTF(("inside process_tmmeasurement_cmd\n"));

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:38 pm | 25/42 |

```

LINE # SOURCE TEXT
2879
2880 reset_obuf();
2881 lm_put_int(TMEASUREMENT_ANS),
2882
2883 inst_id = lm_get_short();
2884 on_or_off = lm_get_char();
2885
2886 /* verify inst_id */
2887 if (inst_id >= user->inst_table_size) {
2888     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: %d specified",
2889         inst_id);
2890     end_queue_message();
2891     end_put(user->id);
2892     return;
2893 }
2894
2895 instance = user->instance[inst_id];
2896 if (BOGUS_INSTANCE(user, instance)) {
2897     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: %d specified",
2898         inst_id);
2899     end_queue_message();
2900     end_put(user->id);
2901     return;
2902 }
2903
2904 switch (on_or_off) {
2905     case LM_TIMING_OFF:
2906         instance->enable_timing_meas = FALSE;
2907         break;
2908     case LM_TIMING_ON:
2909         def_ptr = instance->definition;
2910         if (def_ptr->tim_meas == TRUE) {
2911             lm_queue_message(ERROR_MSG, "Timing Measurement is inhibited for device: %s\n",
2912                 def_ptr->device_name);
2913         }
2914         else {
2915             if (def_ptr->device_type == PUBLIC)
2916                 instance->enable_timing_meas = TRUE;
2917             else {
2918                 if ((instance->has_history == TRUE) &&
2919                     (instance->purge_ptrn_on_next_eval == FALSE)) {
2920                     instance->enable_timing_meas = TRUE;
2921                 }
2922                 else {
2923                     lm_queue_message(ERROR_MSG, "private instance: %s does not have history",
2924                         instance->device_info_string);
2925                 }
2926             }
2927         }
2928         break;
2929     default:
2930         lm_queue_message(ERROR_MSG, "illegal attribute: %d",
2931             on_or_off);
2932         break;
2933 }
2934
2935 end_queue_message();
2936 end_put(user->id);
2937
2938 void process_loop_ptrn_cmd(user)
2939 USER_INFO *user;
2940 {
2941     INSTANCE_INFO *instance;
2942     DEVICE_SPEC *def_ptr;
2943     u_short inst_id;
2944     extern char *modeler_state;
2945
2946     DPRINTF(("inside process_loop_ptrn_cmd\n"));
2947
2948     reset_obuf();
2949     lm_put_int(LOOP_PTRN_ANS);
2950
2951     inst_id = lm_get_short();
2952
2953     /* verify inst_id */
2954     if (inst_id >= user->inst_table_size) {
2955         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
2956             inst_id);
2957         end_queue_message();
2958         end_put(user->id);
2959         return;
2960     }
2961
2962     instance = user->instance[inst_id];
2963     if (BOGUS_INSTANCE(user, instance)) {
2964         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
2965             inst_id);
2966         end_queue_message();
2967         end_put(user->id);
2968         return;
2969     }
2970
2971     def_ptr = instance->definition;
2972     if (def_ptr == NULL) {
2973         lm_queue_message(ERROR_MSG, "internal error: no definition for instance");
2974         end_queue_message();
2975         end_put(user->id);
2976         return;
2977     }
2978
2979     if (((EXTRA_DEVICE_SPEC *)def_ptr->extra_data)->dab_ok == FALSE) {
2980         lm_queue_message(ERROR_MSG, "Device Adapter was Removed");
2981         end_queue_message();
2982         end_put(user->id);
2983         return;
2984     }
2985
2986     if (instance->failed_to_alloc_ptrn == TRUE) {
2987         lm_queue_message(ERROR_MSG, "out of Fast Pattern Memory: cannot continue simulation");
2988         end_queue_message();
2989         end_put(user->id);
2990         return;
2991     }
2992
2993     if (instance->fatal_error == TRUE) {
2994         lm_queue_message(ERROR_MSG, "fatal error encountered on this instance: cannot continue simulation");
2995         end_queue_message();
2996         end_put(user->id);
2997     }
2998 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:38 pm | 26/43 |

```

LINE #          SOURCE TEXT
2999      }
3000      return;
3001      }
3002      if (instance->definition->device_type == PRIVATE) {
3003          lm_queue_message(ERROR_MSG, "Cannot loop pattern for PRIVATE device");
3004          end_queue_message();
3005          end_put(user->id);
3006          return;
3007      }
3008      }
3009      if (active_user_count() > 1) {
3010          lm_queue_message(ERROR_MSG, "modeler busy");
3011          end_queue_message();
3012          end_put(user->id);
3013          return;
3014      }
3015      modeler_state = RUNNING_LOOPMODE;
3016      end_queue_message();
3017      end_put(user->id); /* ack now */
3018      (void)do_loop_ptrn(instance);
3019      modeler_state = MODELER_RUNNING;
3020      }
3021      }
3022      }
3023      }
3024      }
3025      active_user_count()
3026      {
3027          int i, count;
3028          for (i = count = 0; i < MAX_USER_COUNT; ++i)
3029              if (user_info_array[i]->active == TRUE)
3030                  ++count;
3031          return(count);
3032      }
3033      }
3034      }
3035      }
3036      }
3037      void process_reset_inst_cmd(user)
3038      USER_INFO *user;
3039      {
3040          DEVICE_SPEC *def_ptr;
3041          INSTANCE_INFO *instance;
3042          DAB_INFO *dab_ptr;
3043          long inst_id;
3044          u_short error_count;
3045          u_short warning_count;
3046          DPRINTF(("inside process_reset_inst_cmd\n"));
3047          reset_cmd();
3048          lm_put_inst(RESET_INST_AWS);
3049          inst_id = lm_get_short();
3050          /* verify the inst_id */
3051          if ((inst_id < 0) || (inst_id >= user->inst_table_size)) {
3052              lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: id specified");
3053              end_queue_message();
3054              end_put(user->id);
3055              return;
3056          }
3057          instance = user->instance[inst_id];
3058          if (instance->instance_id == 0) {
3059              lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: id specified");
3060              end_queue_message();
3061              end_put(user->id);
3062              return;
3063          }
3064          if (instance->definition->device_type == PRIVATE) {
3065              lm_queue_message(ERROR_MSG, "Device Adapter was removed");
3066              end_queue_message();
3067              end_put(user->id);
3068              return;
3069          }
3070          if (instance->is_fault) {
3071              lm_queue_message(ERROR_MSG, "internal simulator error: cannot reset a fault; fault id: id");
3072              end_queue_message();
3073              end_put(user->id);
3074              return;
3075          }
3076          /* Check if this instance still has faults */
3077          /* ??? Can't do this with the current structure.
3078          /* Just make sure this condition is checked on the host.
3079          */
3080          return_all_ptrn_block(instance);
3081          dab_ptr = dab_list(instance->dab_info_index);
3082          reinitialize_instance(instance);
3083          build_static_pattern_seq(instance);
3084          lm_message_types(error_count, warning_count);
3085          if (error_count != 0) {
3086              dab_ptr->used_as_private = FALSE;
3087              return_all_ptrn_block(instance);
3088              end_queue_message();
3089          }
3090          else {
3091              if (def_ptr->device_type == PRIVATE) {
3092                  instance->has_history = TRUE;
3093                  instance->purge_ptrn_on_next_eval = TRUE;
3094              }
3095              end_queue_message();
3096              copy_initial_values(instance);
3097          }
3098      }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89
TIME 6:14:38 pm

PAGE #
27/44

```

LINE # SOURCE TEXT
3119 }
3120
3121     end_put(user->fd);
3122 }
3123
3124 /* ARGSUSED */
3125 void process_no_such(user)
3126 USER_INFO *user;
3127 {
3128     DPRINTF(("inside process_no_such\n"));
3129
3130     reset_obuf();
3131
3132     lm_put_int(NO_SUCH_ANS);
3133     end_put(user->fd);
3134 }
3135
3136 void process_test_network_cmd(user)
3137 USER_INFO *user;
3138 {
3139     USER_INFO *userx;
3140     int j;
3141     unsigned total_number;
3142     unsigned checksum;
3143     unsigned checksum2;
3144     unsigned command;
3145     unsigned subcommand;
3146     unsigned usernr;
3147     char error_string[512];
3148     unsigned temp;
3149     char c;
3150
3151     DPRINTF(("inside process_test_network\n"));
3152
3153     reset_obuf();
3154
3155     lm_put_int(TEST_NETWORK_ANS);
3156
3157     command = lm_get_int();
3158     switch (command) {
3159     case 1:
3160         total_number = lm_get_int();
3161
3162         DPRINTF(("total_number received = %d\n", total_number));
3163
3164         checksum = 0;
3165         for (j = 0; j < total_number; ++j) {
3166             checksum += lm_get_int();
3167         }
3168
3169         DPRINTF(("checksum on command: %d\n", checksum));
3170
3171         if ((checksum2 = lm_get_int()) != checksum) {
3172             DPRINTF(("Checksum error\n"));
3173             lm_queue_message(ERROR_MSG, "internal error: checksum error on command: exp: %08x got: %08x",
3174                             checksum2, checksum);
3175             end_queue_message();
3176             end_put(user->fd);
3177             return;
3178         }
3179
3180         end_queue_message();
3181
3182         lm_put_int(total_number);
3183
3184         checksum = 0;
3185         for (j = 0; j < total_number; ++j) {
3186             lm_put_int(j*123);
3187             checksum += j * 123;
3188         }
3189
3190         DPRINTF(("checksum on answer: %d\n", checksum));
3191
3192         lm_put_int(checksum);
3193
3194         break;
3195     case 2:
3196         subcommand = lm_get_int();
3197         if (subcommand == 1000) {
3198             /* lock command */
3199             if (modeler_is_locked == TRUE) {
3200                 lm_queue_message(ERROR_MSG, "modeler is currently locked by: %s",
3201                                 user_with_lock);
3202                 end_queue_message();
3203                 end_put(user->fd);
3204                 return;
3205             }
3206             else {
3207                 for (j = 0; (c = lm_get_char()) != '\0'; ++j)
3208                     user_with_lock[j] = c;
3209                 user_with_lock[j] = '\0';
3210                 modeler_is_locked = TRUE;
3211             }
3212         }
3213         else if (subcommand == 2000) {
3214             /* unlock command */
3215             if (modeler_is_locked == TRUE) {
3216                 if (strcmp(user_with_lock, user->username) != 0) {
3217                     lm_queue_message(ERROR_MSG, "failed to unlock modeler, user: %s holds the lock",
3218                                     user_with_lock);
3219                     end_queue_message();
3220                     end_put(user->fd);
3221                     return;
3222                 }
3223                 modeler_is_locked = FALSE;
3224             }
3225             else {
3226                 lm_queue_message(ERROR_MSG, "unknown lock/unlock command");
3227                 end_queue_message();
3228                 end_put(user->fd);
3229                 return;
3230             }
3231         }
3232         break;
3233     case 3:
3234         usernr = lm_get_int();
3235
3236         if (usernr >= MAX_USER_COUNT) {
3237             lm_queue_message(ERROR_MSG, "user: %d too large", usernr);
3238             end_queue_message();

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89
TIME 6:14:38 pm

PAGE #
28/45

```

LINE #      SOURCE TEXT
3239      end_put(user->fd);
3240      return;
3241      }
3242      userx = *****_info_array(userno);
3243      if (userx == NULL) {
3244          lm_queue_message(ERROR_MSG, "user number: %d does not exist", userno);
3245          lm_queue_message();
3246          end_put(user->fd);
3247          return;
3248      }
3249      abort_user(userx);
3250      end_queue_message();
3251      end_put(user->fd);
3252      if (set_close_connection_for_server(table_of_coans[userno]) != SUCCESS) {
3253          while (1) {
3254              if (lm_dequeue_message(&temp, error_string) != FAILURE)
3255                  DEPRINTF(("ts", error_string));
3256              else
3257                  break;
3258          }
3259          return;
3260      }
3261      /* Clear profile counter */
3262      profile_clear();
3263      break;
3264      case 5:
3265          /* Dump profile information */
3266          end_queue_message();
3267          profile_dump_count();
3268          break;
3269      default:
3270          lm_queue_message(ERROR_MSG, "unknown test network command");
3271          end_queue_message();
3272          break;
3273      }
3274      end_put(user->fd);
3275      return;
3276      }
3277      void process_abort_cmd(user)
3278      USER_INFO *user;
3279      {
3280          u_short temp;
3281          char error_string[512];
3282          DEPRINTF(("inside process_abort_cmd\n"));
3283          reset_obuf();
3284          lm_put_int(ABORT_ANS);
3285          abort_user(user);
3286          end_queue_message();
3287          DEPRINTF(("exiting process_abort_cmd; user: %d\n", lm_global_conn_ptr->fd));
3288          end_put(user->fd);
3289          if (set_close_connection_for_server(lm_global_conn_ptr) != SUCCESS) {
3290              while (1) {
3291                  if (lm_dequeue_message(&temp, error_string) != FAILURE)
3292                      DEPRINTF(("ts", error_string));
3293                  else
3294                      break;
3295              }
3296          }
3297      }
3298      /* ARGSUSED */
3299      void process_label_dab_cmd(user)
3300      USER_INFO *user;
3301      {
3302          DAB_EEPROM dab_eeprom;
3303          u_long configuration;
3304          u_char lane0;
3305          u_char slotno;
3306          char dab_type[MAX_STRING_LENGTH];
3307          char device_name[MAX_STRING_LENGTH];
3308          char model_maker[MAX_STRING_LENGTH];
3309          char model_revision[MAX_STRING_LENGTH];
3310          char manufacturer[MAX_STRING_LENGTH];
3311          char revision[MAX_STRING_LENGTH];
3312          u_char i;
3313          u_char j;
3314          char c;
3315          u_char lanes_high;
3316          u_char slots_wide;
3317          u_char segment_number;
3318          DAB_INFO *dab_ptr;
3319          u_char dabno;
3320          u_char unitno;
3321          DEPRINTF(("inside process_label_dab_cmd\n"));
3322          reset_obuf();
3323          lm_put_int(LABEL_DAB_ANS);
3324          lane0 = lm_get_char();
3325          if (lane0 >= MAX_LANE_COUNT) {
3326              lm_queue_message(ERROR_MSG, "illegal lane: %d", lane0);
3327              end_queue_message();
3328              end_put(user->fd);
3329              return;
3330          }
3331          slotno = lm_get_char();
3332          if (slotno >= MAX_SLOT_COUNT) {
3333              lm_queue_message(ERROR_MSG, "illegal slot: %d", slotno);
3334              end_queue_message();
3335              end_put(user->fd);
3336          }

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:38 pm | 29/46 |

| LINE # | SOURCE TEXT |
|--------|--|
| 3359 | return; |
| 3360 | } |
| 3361 | |
| 3362 | /* dab_type */ |
| 3363 | for (i = 0; ((c = lm_get_char()) != '\0'); ++i) |
| 3364 | dab_type[i] = c; |
| 3365 | dab_type[i] = '\0'; |
| 3366 | |
| 3367 | if (strlen(dab_type) > TYPE_LENGTH) { |
| 3368 | lm_queue_message(ERROR_MSG, "Device Adapter type: %s too long; max allowed: %d chars", |
| 3369 | dab_type, TYPE_LENGTH); |
| 3370 | end_queue_message(); |
| 3371 | end_put(user->id); |
| 3372 | return; |
| 3373 | } |
| 3374 | |
| 3375 | /* device_name */ |
| 3376 | for (i = 0; ((c = lm_get_char()) != '\0'); ++i) |
| 3377 | device_name[i] = c; |
| 3378 | device_name[i] = '\0'; |
| 3379 | |
| 3380 | if (strlen(device_name) > NAME_LENGTH) { |
| 3381 | lm_queue_message(ERROR_MSG, "device_name: %s too long; max allowed: %d chars", |
| 3382 | device_name, NAME_LENGTH); |
| 3383 | end_queue_message(); |
| 3384 | end_put(user->id); |
| 3385 | return; |
| 3386 | } |
| 3387 | |
| 3388 | /* model_maker */ |
| 3389 | for (i = 0; ((c = lm_get_char()) != '\0'); ++i) |
| 3390 | model_maker[i] = c; |
| 3391 | model_maker[i] = '\0'; |
| 3392 | |
| 3393 | if (strlen(model_maker) > MAKER_LENGTH) { |
| 3394 | lm_queue_message(ERROR_MSG, "Device Adapter model_maker: %s too long; max allowed: %d chars", |
| 3395 | model_maker, MAKER_LENGTH); |
| 3396 | end_queue_message(); |
| 3397 | end_put(user->id); |
| 3398 | return; |
| 3399 | } |
| 3400 | |
| 3401 | /* model_revision */ |
| 3402 | for (i = 0; ((c = lm_get_char()) != '\0'); ++i) |
| 3403 | model_revision[i] = c; |
| 3404 | model_revision[i] = '\0'; |
| 3405 | |
| 3406 | if (strlen(model_revision) > REV_LENGTH) { |
| 3407 | lm_queue_message(ERROR_MSG, "Device Adapter model_revision: %s too long; max allowed: %d chars", |
| 3408 | model_revision, REV_LENGTH); |
| 3409 | end_queue_message(); |
| 3410 | end_put(user->id); |
| 3411 | return; |
| 3412 | } |
| 3413 | |
| 3414 | /* manufacturer */ |
| 3415 | for (i = 0; ((c = lm_get_char()) != '\0'); ++i) |
| 3416 | manufacturer[i] = c; |
| 3417 | manufacturer[i] = '\0'; |
| 3418 | |
| 3419 | if (strlen(manufacturer) > MAN_LENGTH) { |
| 3420 | lm_queue_message(ERROR_MSG, "Device Adapter manufacturer: %s too long; max allowed: %d chars", |
| 3421 | manufacturer, MAN_LENGTH); |
| 3422 | end_queue_message(); |
| 3423 | end_put(user->id); |
| 3424 | return; |
| 3425 | } |
| 3426 | |
| 3427 | /* revision */ |
| 3428 | for (i = 0; ((c = lm_get_char()) != '\0'); ++i) |
| 3429 | revision[i] = c; |
| 3430 | revision[i] = '\0'; |
| 3431 | |
| 3432 | if (strlen(revision) > REV_LENGTH) { |
| 3433 | lm_queue_message(ERROR_MSG, "Device Adapter revision: %s too long; max allowed: %d chars", |
| 3434 | revision, REV_LENGTH); |
| 3435 | end_queue_message(); |
| 3436 | end_put(user->id); |
| 3437 | return; |
| 3438 | } |
| 3439 | |
| 3440 | lanes_high = lm_get_char(); |
| 3441 | if (lanes_high > MAX_LANE_COUNT) { |
| 3442 | lm_queue_message(ERROR_MSG, "illegal lanes_high: %d; max allowed: %d", |
| 3443 | lanes_high, MAX_LANE_COUNT); |
| 3444 | end_queue_message(); |
| 3445 | end_put(user->id); |
| 3446 | return; |
| 3447 | } |
| 3448 | |
| 3449 | if (lanes_high == 0) { |
| 3450 | lm_queue_message(ERROR_MSG, "illegal lanes_high: 0; min allowed: 1"); |
| 3451 | end_queue_message(); |
| 3452 | end_put(user->id); |
| 3453 | return; |
| 3454 | } |
| 3455 | |
| 3456 | slots_wide = lm_get_char(); |
| 3457 | if (slots_wide > MAX_SLOT_COUNT) { |
| 3458 | lm_queue_message(ERROR_MSG, "invalid slots_wide: %d; max allowed: %d", |
| 3459 | slots_wide, MAX_SLOT_COUNT); |
| 3460 | end_queue_message(); |
| 3461 | end_put(user->id); |
| 3462 | return; |
| 3463 | } |
| 3464 | |
| 3465 | if (slots_wide == 0) { |
| 3466 | lm_queue_message(ERROR_MSG, "invalid slots_wide: 0; min allowed: 1"); |
| 3467 | end_queue_message(); |
| 3468 | end_put(user->id); |
| 3469 | return; |
| 3470 | } |
| 3471 | |
| 3472 | segment_number = lm_get_char(); |
| 3473 | if (segment_number > MAX_SEGMENT_PER_DEVICE) { |
| 3474 | lm_queue_message(ERROR_MSG, "invalid segment_number: %d; max allowed: %d", |
| 3475 | segment_number, MAX_SEGMENT_PER_DEVICE); |
| 3476 | end_queue_message(); |
| 3477 | end_put(user->id); |
| 3478 | return; |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89
TIME 6:14:38 pm

PAGE #
30/47

```

LINE # SOURCE TEXT
3479 }
3480
3481 if (check_dab_present(laneno, slotno) == FAILURE) {
3482     lm_queue_message(ERROR_MSG, "no Device Adapter in lane: %c slot: %d",
3483         'A' + laneno, slotno);
3484     end_queue_message();
3485     end_put(user->id);
3486     return;
3487 }
3488
3489 /* Check through the dab list and see if any active instances or faults
3490 exit for this position.
3491 */
3492 for (dabno = 0; dabno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++dabno) {
3493     dab_ptr = dab_list[dabno];
3494     if (dab_ptr == NULL)
3495         continue;
3496
3497     for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
3498         if ((dab_ptr->unit_location[unitno].lane_no == laneno) &&
3499             (dab_ptr->unit_location[unitno].slot_no == slotno)) {
3500             if ((dab_ptr->act_inst_count + dab_ptr->act_var_count) > 0) {
3501                 lm_queue_message(ERROR_MSG, "Device Adapter in lane: %c slot: %d is in use.",
3502                     'A' + laneno, slotno);
3503                 end_queue_message();
3504                 end_put(user->id);
3505                 return;
3506             }
3507         }
3508     }
3509 }
3510
3511 /* Get the insertion count into the dab_eeprom structure */
3512 (void)lm_read_eeprom(laneno * MAX_SLOT_COUNT + slotno, &dab_eeprom);
3513
3514 for (i = 0; device_name[i] != '\0'; ++i)
3515     dab_eeprom.device_name[i] = device_name[i];
3516
3517 for (; i < NAME_LENGTH; ++i)
3518     dab_eeprom.device_name[i] = '\0';
3519
3520 for (i = 0; model_maker[i] != '\0'; ++i)
3521     dab_eeprom.model_maker[i] = model_maker[i];
3522
3523 for (; i < MAKER_LENGTH; ++i)
3524     dab_eeprom.model_maker[i] = '\0';
3525
3526 for (i = 0; model_revision[i] != '\0'; ++i)
3527     dab_eeprom.model_revision[i] = model_revision[i];
3528
3529 for (; i < REV_LENGTH; ++i)
3530     dab_eeprom.model_revision[i] = '\0';
3531
3532 for (i = 0; manufacturer[i] != '\0'; ++i)
3533     dab_eeprom.dab_manufacturer[i] = manufacturer[i];
3534
3535 for (; i < MAN_LENGTH; ++i)
3536     dab_eeprom.dab_manufacturer[i] = '\0';
3537
3538 for (i = 0; revision[i] != '\0'; ++i)
3539     dab_eeprom.dab_revision[i] = revision[i];
3540
3541 for (; i < REV_LENGTH; ++i)
3542     dab_eeprom.dab_revision[i] = '\0';
3543
3544 for (i = 0; dab_type[i] != '\0'; ++i)
3545     dab_eeprom.dab_type[i] = dab_type[i];
3546
3547 for (; i < TYPE_LENGTH; ++i)
3548     dab_eeprom.dab_type[i] = '\0';
3549
3550 dab_eeprom.segment_number = segment_number;
3551
3552 configuration = 0;
3553 for (i = 0; i < lanes_high; ++i) {
3554     for (j = 0; j < slots_wide; ++j) {
3555         configuration |= 1 << (i * MAX_SLOT_COUNT + j);
3556     }
3557 }
3558
3559 dab_eeprom.configuration = configuration;
3560
3561 /* Write the DAB EEPROM structure to the EEPROM on the DAB */
3562 (void)lm_write_eeprom(laneno * MAX_SLOT_COUNT + slotno, &dab_eeprom);
3563
3564 reconfigure_dab();
3565
3566 end_queue_message();
3567 end_put(user->id);
3568 }
3569
3570 void process_eval_control_cmd(user)
3571 USER_INFO *user;
3572 {
3573     INSTANCE_INFO *instance;
3574     u_short inst_id;
3575     u_short attrib;
3576     long value;
3577
3578     DPRINTF(("inside process_eval_control_cmd\n"));
3579
3580     reset_obuf();
3581     lm_put_inst(EVAL_CONTROL_ANS);
3582
3583     inst_id = lm_get_short();
3584     attrib = lm_get_inst();
3585     value = lm_get_inst();
3586
3587     /* verify inst_id */
3588     if (inst_id >= user->inst_table_size) {
3589         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
3590             inst_id);
3591         end_queue_message();
3592         end_put(user->id);
3593         return;
3594     }
3595
3596     instance = user->instance[inst_id];
3597     if (IS_OGUS_INSTANCE(user, instance)) {
3598         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/function.c

DATE 5/23/89
TIME 6:14:38 pm

PAGE #
31/48

```

LINE #          SOURCE TEXT
3599             inst_id);
3600             end_queue_message();
3601             end_put(user->id);
3602             return;
3603         }
3604     }
3605     switch (attrib) {
3606     case LM_NUMBER_OF_SAMPLES:
3607         instance->sample_count = value;
3608         break;
3609     case LM_NUMBER_OF_EVALUATIONS:
3610         if (value == LM EVERY_EVALUATION)
3611             instance->evaluation_count = MAXINT;
3612         else
3613             instance->evaluation_count = value;
3614         break;
3615     default:
3616         lm_queue_message(ERROR_MSG, "illegal attribute: id",
3617             attrib);
3618         break;
3619     }
3620     end_queue_message();
3621     end_put(user->id);
3622 }
3623
3624 void process_reboot_cmd(user)
3625 USER_INFO *user;
3626 {
3627     DPRINTF(("inside process_reboot_cmd\n"));
3628     reset_obuf();
3629     lm_put_int(REBOOT_ANS);
3630
3631 #ifdef MODELER
3632     if (check_password(user) == TRUE) {
3633         reboot_flag = lm_get_char();
3634         reboot_delay_time = lm_get_int() * 1000, /* ms */
3635         time_reboot_was_issued = lm_time();
3636         rebooting = TRUE;
3637     }
3638     else {
3639         lm_queue_message(ERROR_MSG, "Password required to reboot modeler");
3640     }
3641 #else
3642     lm_queue_message(ERROR_MSG, "Core Modeler Code is not running on modeler; command ignored...");
3643 #endif
3644     end_queue_message();
3645     end_put(user->id);
3646 }
3647
3648 void process_shutdown_cmd(user)
3649 USER_INFO *user;
3650 {
3651     DPRINTF(("inside process_shutdown_cmd\n"));
3652     reset_obuf();
3653     lm_put_int(SHUTDOWN_ANS);
3654
3655 #ifdef MODELER
3656     if (check_password(user) == TRUE) {
3657         reboot_flag = lm_get_char();
3658         reboot_delay_time = lm_get_int() * 1000, /* ms */
3659         time_reboot_was_issued = lm_time();
3660         rebooting = TRUE;
3661         shutdown = TRUE;
3662     }
3663     else {
3664         lm_queue_message(ERROR_MSG, "Password required to shutdown modeler");
3665     }
3666 #else
3667     lm_queue_message(ERROR_MSG, "Core Modeler Code is not running on modeler; command ignored...");
3668 #endif
3669     end_queue_message();
3670     end_put(user->id);
3671 }
3672
3673 void process_password_cmd(user)
3674 USER_INFO *user;
3675 {
3676     u_short attrib;
3677     char password[MAX_STRING_LENGTH];
3678     char crypt_password[CRYPTED_PASSWORD_LENGTH];
3679     short i;
3680
3681     DPRINTF(("inside process_password_cmd\n"));
3682     reset_obuf();
3683     lm_put_int(PASSWORD_ANS);
3684     attrib = lm_get_int();
3685
3686     i = 0;
3687     while (password[i++] != lm_get_char())
3688         ;
3689
3690     lm_crypt(password, crypt_password);
3691
3692     switch (attrib) {
3693     case LM_ASSIGN:
3694         if (check_password(user) == TRUE) {
3695             if (write_password(crypt_password) == SUCCESS) {
3696                 for (i = 0; i < MAX_USER_COUNT; ++i) {
3697                     user_info_array[i]->good_password = FALSE;
3698                 }
3699                 user->good_password = TRUE;
3700             }
3701             else {
3702                 lm_queue_message(ERROR_MSG, "Password required to set new password on modeler");
3703             }
3704             break;
3705         case LM_ENTER:
3706             user->good_password = compare_password(crypt_password);
3707         }
3708     }

```

| | | | | | |
|--|--|-------------------------------------|--|--------------------|-----------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/function.c | | DATE 5/23/89 | PAGE # 32/49 |
| | | | | TIME 6:14:38 pm | |
| LINE # | SOURCE TEXT | | | | |
| 3719 | break; | | | | |
| 3720 | default: | | | | |
| 3721 | lm_queue_message(ERROR_MSG, "illegal attribute: %d", | | | | |
| 3722 | attrib); | | | | |
| 3723 | break; | | | | |
| 3724 | } | | | | |
| 3725 | | | | | |
| 3726 | end_queue_message(); | | | | |
| 3727 | end_put(user->fd); | | | | |
| 3728 | | | | | |
| 3729 | | | | | |
| 3730 | | | | | |

1421

5,353,243

1422

Copyright 1989

Logic Modeling Systems

HEADER FILE

lm1000/function.h

DATE

5/23/89

PAGE #

TIME

6:14:41 pm

1/50

LINE # HEADER TEXT

```
1 /* SCCS_ID: function.h rev 3.1, 4/24/89 at 07:52:59 */
2
3 extern void process_create_def_cmd();
4 extern void process_create_instance_cmd();
5 extern void process_create_fault_cmd();
6
7 extern void process_release_def_cmd();
8 extern void process_release_instance_cmd();
9 extern void process_release_fault_cmd();
10
11 extern void process_eval_cmd();
12
13 extern void process_save_def_cmd();
14 extern void process_save_def_cont_cmd();
15 extern void process_save_ptrn_cmd();
16 extern void process_save_ptrn_cont_cmd();
17 extern void process_restore_inst_cmd();
18 extern void process_restore_ptrn_cmd();
19 extern void process_ptrn_hist_cmd();
20
21 extern void process_inq_modeler_cmd();
22 extern void process_inq_user_list_cmd();
23 extern void process_inq_user_cmd();
24 extern void process_inq_lane_cmd();
25 extern void process_inq_pam_cmd();
26 extern void process_inq_pai_cmd();
27 extern void process_inq_device_list_cmd();
28 extern void process_inq_device_name_cmd();
29 extern void process_inq_device_cmd();
30 extern void process_inq_dab_cmd();
31 extern void process_inq_dab_loc_cmd();
32 extern void process_inq_instance_cmd();
33 extern void process_inq_fault_cmd();
34
35 extern void process_inq_avail_ptrn_cmd();
36
37 extern void process_tmeasuremnt_cmd();
38
39 extern void process_loop_ptrn_cmd();
40
41 extern void process_reset_inst_cmd();
42
43 extern void process_test_network_cmd();
44
45 extern void process_abort_cmd();
46
47 extern void process_begin_session_cmd();
48
49 extern void process_label_dab_cmd();
50
51 extern void process_eval_control_cmd();
52
53 extern void process_reboot_cmd();
54 extern void process_shutdown_cmd();
55
56 extern void process_check_dabdef_cmd();
57
58 extern void process_password_cmd();
59
60 extern void process_lm_read();
61 extern void process_lm_write();
62
63 extern void process_no_such();
```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hardware.c

DATE 5/23/89
TIME 6:14:41 pm

PAGE #
1/51

```

LINE # SOURCE TEXT
1  /* SCCS_ID: hardware.c rev 3.1, 4/24/89 at 07:53:02 */
2
3  #include "device.h"
4  #include "hardware.h"
5  #include "eeprom.h"
6  #include "laser.h"
7  #include "id.h"
8
9  #define LONG 0
10 #define SHORT 1
11 #define CHAR 2
12
13 extern void id_load();
14
15 typedef struct {
16     DAB_EEPROM dab_eeprom;
17     u_long active;
18 } DAB_EEPROM_INFO;
19
20 DAB_EEPROM_INFO dab_eeprom_info[MAX_LANE_COUNT][MAX_SLOT_COUNT];
21
22 /*-----*/
23
24 init_dab_eeprom_info()
25 {
26     u_char lanes;
27     u_char slots;
28
29     for (lanes = 0; lanes < MAX_LANE_COUNT; ++lanes) {
30         for (slots = 0; slots < MAX_SLOT_COUNT; ++slots) {
31             dab_eeprom_info[lanes][slots].active = FALSE;
32         }
33     }
34 }
35
36 u_char
37 read_loc_char(address)
38 u_char *address,
39 {
40
41     #ifdef MODELER
42         return(*address);
43     #else
44     #ifdef DIRECTCONN
45         u_long value;
46
47         dc_read_loc(CHAR, (u_long)address, &value);
48
49         return(value & 0xff);
50     #else
51         u_long value;
52
53         (void)db_fetch((u_long)address, &value);
54
55         return(value);
56     #endif
57     #endif
58 }
59
60 /*-----*/
61
62 u_short
63 read_loc_short(address)
64 u_short *address,
65 {
66
67     #ifdef MODELER
68         return(*address);
69     #else
70     #ifdef DIRECTCONN
71         u_long value;
72
73         dc_read_loc(SHORT, (u_long)address, &value);
74
75         return(value & 0xffff);
76     #else
77         u_long value;
78
79         (void)db_fetch((u_long)address, &value);
80
81         return(value);
82     #endif
83     #endif
84 }
85
86 /*-----*/
87
88 u_long
89 read_loc_long(address)
90 u_long *address,
91 {
92     /*
93      * returns the contents of the location at address
94      */
95     #ifdef MODELER
96         return(*address);
97     #else
98     #ifdef DIRECTCONN
99         u_long value;
100
101         dc_read_loc(LONG, (u_long)address, &value);
102
103         return(value);
104     #else
105         u_long value;
106
107         (void)db_fetch((u_long)address, &value);
108
109         return(value);
110     #endif
111     #endif
112 }
113
114 /*-----*/
115
116 void
117 write_loc_char(address, value)
118 u_char *address,
119 u_char value;
120

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hardware.c

DATE 5/23/89
TIME 6:14:41 pm

PAGE #
2/52

```

LINE #          SOURCE TEXT
121  {
122  /*
123   * write value into location address
124   */
125  }
126  #ifdef MODELER
127  *address = value;
128  #else
129  #ifdef DIRECTCONN
130  dc_write_loc(CHAR, (u_long)address, value);
131  #else
132  (void)db_write((u_long)address, value);
133  #endif
134  #endif
135  }
136  void
137  write_loc_short(address, value)
138  {
139  u_short *address;
140  u_short value;
141  {
142  /*
143   * write value into location address
144   */
145  }
146  #ifdef MODELER
147  *address = value;
148  #else
149  #ifdef DIRECTCONN
150  dc_write_loc(SHORT, (u_long)address, value);
151  #else
152  (void)db_write((u_long)address, value);
153  #endif
154  #endif
155  }
156  void
157  write_loc_long(address, value)
158  {
159  u_long *address;
160  u_long value;
161  {
162  /*
163   * write value into location address
164   */
165  }
166  #ifdef MODELER
167  *address = value;
168  #else
169  #ifdef DIRECTCONN
170  dc_write_loc(LONG, (u_long)address, value);
171  #else
172  (void)db_write((u_long)address, value);
173  #endif
174  #endif
175  }
176  /*-----*/
177  probe(addr)
178  {
179  u_long addr;
180  {
181  #ifdef MODELER
182  return(lm_read_probe(addr));
183  #else
184  #ifdef DIRECTCONN
185  return(dc_probe(addr));
186  #else
187  u_long junk;
188  return(db_fetch(addr, &junk));
189  #endif
190  #endif
191  }
192  /*-----*/
193  read_cpu(tot_system_mem, slot_count, lane_count)
194  {
195  u_long *tot_system_mem;
196  u_char *slot_count;
197  u_char *lane_count;
198  {
199  /*
200   * return the tot_system_mem, and clab_board_id.
201   * This function better not fail !!
202   */
203  ID_PROM_CPU cpu_id_prom;
204  u_long status;
205  u_short model_number;
206  u_char dram_size;
207  #ifdef MODELER
208  id_load((u_char *)CPU_ID_PROM_ADDR, (char *)&cpu_id_prom);
209  dram_size = cpu_id_prom.dram_size;
210  *tot_system_mem = (dram_size + 1) * ID_CPU_DRAM_SIZE_K;
211  model_number = cpu_id_prom.model_number;
212  if (model_number == 1000) {
213  *slot_count = 8;
214  *lane_count = 4;
215  }
216  return(SUCCESS);
217  #else
218  #ifdef DIRECTCONN
219  *tot_system_mem = 4 * ID_CPU_DRAM_SIZE_K;
220  model_number = 1000;
221  if (model_number == 1000) {
222  *slot_count = 8;
223  *lane_count = 4;
224  }
225  return(SUCCESS);
226  #else
227  *tot_system_mem = read_loc_long((u_long *)CPU_MEM_SIZE_ADDR);

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/hardware.c | DATE 5/23/89 | PAGE # 3/53 |
|--|--|---|-----------------|----------------|
| LINE # | | SOURCE TEXT | | |
| 241 | | *slot_count = read_loc_long((u_long *)CPU_SLOT_COUNT_ADDR); | | |
| 242 | | *lane_count = read_loc_long((u_long *)CPU_LANE_COUNT_ADDR); | | |
| 243 | | return(SUCCESS); | | |
| 244 | | endif | | |
| 245 | | endif | | |
| 246 | | } | | |
| 247 | | } | | |
| 248 | | } | | |
| 249 | | /*-----*/ | | |
| 250 | | read_pac(lane_number) | | |
| 251 | | u_char lane_number; | | |
| 252 | | { | | |
| 253 | | /* | | |
| 254 | | * return the pac_board_id and the slot_count of the lane_number. | | |
| 255 | | * return 0 if lane does not exist else return 1. | | |
| 256 | | */ | | |
| 257 | | u_long addr; | | |
| 258 | | addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC + | | |
| 259 | | LANE_PAC_START_OFFSET + PAC_BOARD_ID_OFFSET; | | |
| 260 | | { | | |
| 261 | | if (probe(addr) == FAILURE) | | |
| 262 | | return(FAILURE); | | |
| 263 | | return(SUCCESS); | | |
| 264 | | } | | |
| 265 | | } | | |
| 266 | | } | | |
| 267 | | /*-----*/ | | |
| 268 | | read_pel(lane_number, slot_number, pel_board_id) | | |
| 269 | | u_char lane_number; | | |
| 270 | | u_char slot_number; | | |
| 271 | | u_long *pel_board_id; | | |
| 272 | | { | | |
| 273 | | /* | | |
| 274 | | * return the board_id of the PEL on geographical | | |
| 275 | | * address lane_number and slot addr. | | |
| 276 | | * return 0 if PEL does not exist else return 1. | | |
| 277 | | */ | | |
| 278 | | u_long addr; | | |
| 279 | | /* check if PEL exists */ | | |
| 280 | | addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC + | | |
| 281 | | LANE_PEL_0_START_OFFSET + slot_number * LANE_PEL_ADDR_INC + | | |
| 282 | | PEL_ID_FROM_OFFSET; | | |
| 283 | | { | | |
| 284 | | if (probe(addr) == FAILURE) | | |
| 285 | | return(FAILURE); | | |
| 286 | | *pel_board_id = read_loc_long((u_long *)addr); | | |
| 287 | | return(SUCCESS); | | |
| 288 | | } | | |
| 289 | | } | | |
| 290 | | /*-----*/ | | |
| 291 | | read_tmg(tmg_board_id) | | |
| 292 | | u_long *tmg_board_id; | | |
| 293 | | { | | |
| 294 | | /* | | |
| 295 | | * return the board_id of the Timing Generator | | |
| 296 | | */ | | |
| 297 | | u_long addr; | | |
| 298 | | addr = CLOS_START_ADDR + CLOS_BOARD_ID_OFFSET; | | |
| 299 | | { | | |
| 300 | | if (probe(addr) == FAILURE) | | |
| 301 | | return(FAILURE); | | |
| 302 | | *tmg_board_id = read_loc_long((u_long *)addr); | | |
| 303 | | return(SUCCESS); | | |
| 304 | | } | | |
| 305 | | /*-----*/ | | |
| 306 | | check_dab_present(lane_number, slot_number) | | |
| 307 | | u_char lane_number; | | |
| 308 | | u_char slot_number; | | |
| 309 | | { | | |
| 310 | | u_long addr; | | |
| 311 | | u_short word; | | |
| 312 | | { | | |
| 313 | | dab_eeprom_info(lane_number)(slot_number).active = FALSE; | | |
| 314 | | if (dab_present(lane_number, slot_number) == FAILURE) { | | |
| 315 | | return(FAILURE); | | |
| 316 | | } | | |
| 317 | | addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC + | | |
| 318 | | LANE_PEL_0_START_OFFSET + slot_number * LANE_PEL_ADDR_INC + | | |
| 319 | | PEL_STATUS_CONTROL_OFFSET; | | |
| 320 | | { | | |
| 321 | | word = read_loc_long((u_long *)addr); | | |
| 322 | | /* assert RESET with INITIALIZE 0 in anticipation for an | | |
| 323 | | * EEPROM write later. | | |
| 324 | | { | | |
| 325 | | word = PEL_CS_RESET_MASK & PEL_CS_INITIALIZE_MASK; | | |
| 326 | | write_loc_long((u_long *)addr, (u_long)word); | | |
| 327 | | } | | |
| 328 | | return(SUCCESS); | | |
| 329 | | } | | |
| 330 | | /*-----*/ | | |
| 331 | | read_dab(lane_number, slot_number, man_id, dab_type, revision, part_name, | | |
| 332 | | model_maker, model_revision, | | |
| 333 | | dab_shape, dab_segment_number, insertion_count) | | |
| 334 | | { | | |
| 335 | | u_char lane_number; | | |
| 336 | | u_char slot_number; | | |
| 337 | | char *man_id; | | |
| 338 | | char *dab_type; | | |
| 339 | | char *revision; | | |
| 340 | | char *part_name; | | |
| 341 | | char *model_maker; | | |
| 342 | | char *model_revision; | | |
| 343 | | } | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hardware.c

DATE 5/23/89

PAGE #

TIME 6:14:41 pm

4/54

```

LINE # SOURCE TEXT
361 u_long *dab_shape;
362 u_char *dab_segment_number;
363 u_short *insertion_count;
364 {
365 /*
366  * Return the info stored in the EEPROM for the DAB on
367  * geographical address lane_number and slot_addr.
368  * return 0 if DAB does not exist else return 1.
369  */
370
371 #ifdef DBASE
372 EEPROM_CONTENTS *eeprom;
373 #endif
374 DAB_EEPROM dab_eeprom;
375 u_long addr;
376 u_short word;
377 u_char i;
378
379 DPRINTF(("inside read_dab() lane: %d slot: %d\n",
380         lane_number, slot_number));
381
382 if (dab_present(lane_number, slot_number) == FAILURE) {
383     DPRINTF(("dab absent\n"));
384     dab_eeprom_info[lane_number][slot_number].active = FALSE;
385     return(FAILURE);
386 }
387
388 DPRINTF(("dab present\n"));
389
390 #ifdef DBASE
391 addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +
392         LANE_PEL_0_START_OFFSET + slot_number * LANE_PEL_ADDR_INC +
393         PEL_STATUS_CONTROL_OFFSET;
394
395 /* Turn off the 1s use LED */
396 word = read_loc_long((u_long *)addr) & 0xffff & PEL_CS_IN_USE_LED_MASK;
397 write_loc_long((u_long *)addr, (u_long)word);
398
399 DPRINTF(("rd: 1s: %d sl: %d pel status: %04X\n",
400         lane_number, slot_number, word));
401
402 /* If the ACTIVE bit is set then we already know the contents of EEPROM */
403 if (((word & PEL_CS_ACTIVE_MASK) != 0) &&
404     (dab_eeprom_info[lane_number][slot_number].active == TRUE)) {
405     dab_eeprom = dab_eeprom_info[lane_number][slot_number].dab_eeprom;
406 }
407 else {
408     DPRINTF(("calling lm_read_eeprom() lane: %d slot: %d\n",
409             lane_number, slot_number));
410     /* assert RESET */
411     word = PEL_CS_RESET_MASK;
412     write_loc_long((u_long *)addr, (u_long)word);
413
414     if (lm_read_eeprom(lane_number * MAX_SLOT_COUNT + slot_number,
415                        &dab_eeprom) == FAILURE) {
416         init_magic_and_pel(lane_number, slot_number);
417         DPRINTF(("failure in lm_read_eeprom()\n"));
418         dab_eeprom_info[lane_number][slot_number].active = FALSE;
419         return(FAILURE);
420     }
421
422     dab_eeprom_info[lane_number][slot_number].active = TRUE;
423     dab_eeprom_info[lane_number][slot_number].dab_eeprom = dab_eeprom;
424 }
425
426 for (i = 0; i < NAME_LENGTH; ++i)
427     part_name[i] = dab_eeprom.device_name[i];
428 part_name[i] = '\0';
429
430 for (i = 0; i < MAKER_LENGTH; ++i)
431     model_maker[i] = dab_eeprom.model_maker[i];
432 model_maker[i] = '\0';
433
434 for (i = 0; i < REV_LENGTH; ++i)
435     model_revision[i] = dab_eeprom.model_revision[i];
436 model_revision[i] = '\0';
437
438 for (i = 0; i < MAN_LENGTH; ++i)
439     man_id[i] = dab_eeprom.dab_manufacturer[i];
440 man_id[i] = '\0';
441
442 for (i = 0; i < REV_LENGTH; ++i)
443     revision[i] = dab_eeprom.dab_revision[i];
444 revision[i] = '\0';
445
446 for (i = 0; i < TYPE_LENGTH; ++i)
447     dab_type[i] = dab_eeprom.dab_type[i];
448 dab_type[i] = '\0';
449
450 *dab_shape = dab_eeprom.configuration;
451 *dab_segment_number = dab_eeprom.segment_number;
452 *insertion_count = dab_eeprom.insertion_count;
453
454 if (*dab_shape == 0) {
455     init_magic_and_pel(lane_number, slot_number);
456     DPRINTF(("dab shape is 0\n"));
457     dab_eeprom_info[lane_number][slot_number].active = FALSE;
458     return(FAILURE);
459 }
460
461 return(SUCCESS);
462
463 #else
464 addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +
465         LANE_PEL_0_START_OFFSET + slot_number * LANE_PEL_ADDR_INC;
466
467 if (!db_fetch_array(addr + PEL_DAB_EEPROM_OFFSET, (char **)&eeprom))
468     return(FAILURE);
469
470 (void)strcpy(man_id, eeprom->man_id);
471 (void)strcpy(dab_type, eeprom->dab_type);
472 (void)strcpy(revision, eeprom->revision);
473 (void)strcpy(part_name, eeprom->part_name);
474 *dab_shape = eeprom->dab_shape;
475 *dab_segment_number = eeprom->dab_segment_number;
476 *insertion_count = 0;
477 return(SUCCESS);
478 #endif
479 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hardware.c

DATE 5/23/89
TIME 6:14:41 pm

PAGE #
5/55

```

LINE # SOURCE TEXT
481
482 /* ???
483 if (! db_fetch_array(addr + 0x00, (char **) &seeprom))
484     return(FAILURE);
485
486 {void}strcpy(mas_id, seeprom->mas_id);
487 {void}strcpy(dab_type, seeprom->dab_type);
488 {void}strcpy(revname, seeprom->revname);
489 {void}strcpy(part_name, seeprom->part_name);
490 *dab_shape = seeprom->dab_shape;
491 *dab_segment_number = seeprom->dab_segment_number;
492 *insertion_count = 0;
493 return(SUCCESS);
494 }
495
496
497 dab_present(lane_number, slot_number)
498 u_char lane_number;
499 u_char slot_number;
500 {
501     u_long addr;
502     u_long status;
503
504     addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +
505           LANE_PEL_0_START_OFFSET + slot_number * LANE_PEL_ADDR_INC +
506           PEL_STATUS_CONTROL_OFFSET;
507
508     /* PEL does not exist */
509     if (system_config->lane(lane_number)->pel(slot_number) == NULL)
510         return(FAILURE);
511
512     status = read_loc_long((u_long *)addr);
513     DPRINTF(("dp: ls: %d sl: %d pel status: %08x\n",
514             lane_number, slot_number, status));
515
516     if (status & PEL_CS_PRESENT_MASK)
517         return(SUCCESS);
518     return(FAILURE);
519 }
520
521
522
523
524 read_pam_count_reg(lane_number, count)
525 u_char lane_number;
526 u_char *count;
527 {
528     u_long addr;
529     u_long temp;
530
531     *count = 0;
532
533     if (read_pac(lane_number) == FAILURE)
534         return(FAILURE);
535
536     addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +
537           LANE_PAC_START_OFFSET + PAC_PAM_COUNT_REG_OFFSET;
538
539     temp = read_loc_long((u_long *)addr) & 0xf;
540
541     switch (temp) {
542     case PAC_PAM_COUNT_EQ_0:
543         *count = 0;
544         break;
545     case PAC_PAM_COUNT_EQ_1:
546         *count = 1;
547         break;
548     case PAC_PAM_COUNT_EQ_2:
549         *count = 2;
550         break;
551     case PAC_PAM_COUNT_EQ_3:
552         *count = 3;
553         break;
554     case PAC_PAM_COUNT_EQ_4:
555         *count = 4;
556         break;
557     default:
558         *count = 0;
559         break;
560     }
561     return(SUCCESS);
562 }
563
564
565
566 read_pam_present(lane_number, panno, value)
567 u_char lane_number;
568 u_char panno;
569 u_char *value;
570 {
571     /* If PAM present --> return "value" = 1 otherwise return "value" = 0 */
572
573     u_long addr;
574
575     *value = 0;
576     if (read_pac(lane_number) == FAILURE)
577         return(FAILURE);
578
579     addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +
580           LANE_PAC_START_OFFSET + PAC_PAM_0_PRESENT_REG_OFFSET +
581           panno * PAC_PAM_PRESENT_REG_ADDR_INC;
582
583     #ifdef DBASE
584     /* PAM is present if DO == 0 */
585     *value = (read_loc_long((u_long *)addr) & 1) == 0;
586     #else
587     *value = (read_loc_long((u_long *)addr) & 1) == 1;
588     #endif
589
590     return(SUCCESS);
591 }
592
593
594
595 read_pam_id(lane_number, panno, pam_type, pam_board_id)
596 u_char lane_number;
597 u_char panno;
598 u_char *pam_type;
599 u_long *pam_board_id;
600 {

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/hardware.c

DATE

5/23/89

PAGE #

TIME

6:14:41 pm

6/56

| LINE # | SOURCE TEXT |
|--------|--|
| 601 | id_prom_pam pam_id_prom; |
| 602 | u_long addr; |
| 603 | |
| 604 | if (read_pac(lane_number) == FAILURE) |
| 605 | return(FAILURE); |
| 606 | |
| 607 | /* The IDPRON on the PAM is located at the LSB of the longword boundary, |
| 608 | * therefore we need to add 3 to the address. |
| 609 | */ |
| 610 | addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC + |
| 611 | LANE_PAC_START_OFFSET + PAC_PAM_0_ID_SIZE_REG_OFFSET + |
| 612 | pamno * PAC_PAM_ID_SIZE_REG_ADDR_INC + 3; |
| 613 | |
| 614 | #ifdef MODELER |
| 615 | id_load((u_char *)addr, (char *)pam_id_prom); |
| 616 | |
| 617 | switch (pam_id_prom.generic.board_type) { |
| 618 | case ID_BT_PAM128K: |
| 619 | *pam_type = PAC_PAM_128K; |
| 620 | break; |
| 621 | case ID_BT_PAM512K: |
| 622 | *pam_type = PAC_PAM_512K; |
| 623 | break; |
| 624 | case ID_BT_PAM2M: |
| 625 | *pam_type = PAC_PAM_2M; |
| 626 | break; |
| 627 | default: |
| 628 | *pam_type = PAC_PAM_128K; |
| 629 | break; |
| 630 | } |
| 631 | |
| 632 | *pam_board_id = 0; |
| 633 | |
| 634 | else |
| 635 | *pam_type = PAC_PAM_128K; |
| 636 | *pam_board_id = 0; |
| 637 | #endif |
| 638 | |
| 639 | return(SUCCESS); |
| 640 | } |
| 641 | |
| 642 | /*-----*/ |
| 643 | cpu_tag_interrupt_status() |
| 644 | { |
| 645 | u_long status; |
| 646 | |
| 647 | /* This bit is active LOW */ |
| 648 | status = read_loc_long((u_long *)CPU_STATUS_REG_ADDR); |
| 649 | |
| 650 | if (status & CPU_TAG_INTR_MASK) |
| 651 | return(FALSE); |
| 652 | |
| 653 | return(TRUE); |
| 654 | } |
| 655 | |
| 656 | /*-----*/ |
| 657 | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hex.c

DATE 5/23/89
TIME 6:14:41 pm

PAGE #
1/57

```

1  /* SCLS_ID: hex.c rev 3.1. 4/24/89 at 07:53:05 */
2
3  /*
4  ** MAIN.C
5  ** This file contains the startup code for the tasks.
6  ** Tasks and resources begin here
7  */
8  #include "common.h"
9  #include "cpu.h"
10 #include "task.h"
11 #include "lance.h"
12 #include "mod_err.h"
13 #include "sparam.h"
14 #include "lm_rd_wr.h"
15
16 /*
17 ** The globals
18 */
19 BOOT_STRUCT boot;
20 extern unsigned long timer_semaphore, rcv_lance_pkt_semaphore;
21 int play_semaphore, malloc_semaphore;
22
23 ///////////////////////////////////////////////////
24 //
25 // The BSP creates the main task, and looks for the entry point
26 // named main.
27 ///////////////////////////////////////////////////
28 u_long auto_start = 0;
29 main()
30 {
31
32 void housekeeping_task();
33 void output_routine();
34 void tx_task();
35 void receive_task();
36 extern void set_boot();
37 extern u_short lm_sparam_access();
38 void pattern_task();
39 u_short write_lance_car(), read_lance_car();
40 int err;
41 extern u_long modeler_inet;
42 extern
43 unsigned long error;
44
45 /*
46 ** This routine initializes the partition used in the system.
47 ** The only usage is by malloc and friends.
48 */
49 create_malloc_partition(&end, (u_long)CPU_RAM_SIZE - (u_long)&end);
50
51 /*
52 ** The LANCE requires that the STOP bit of CS0 be set
53 ** before accessing CS0, CS2, or CS3.
54 if (write_lance_car(SELECT_CS0, STOP_ACTIVITY) == FAILURE)
55     output_routine("Error trying to set stop ");
56
57 if (lm_sparam_access((char *) &boot, BOOT, sizeof BOOT, MEMORY_READ, &err) == FAILURE)
58     output_routine("Failed to read NParam");
59
60 modeler_inet = boot.modeler_internet_address;
61
62 /*
63 ** Initialize the fifos
64 */
65 fifo_initialize();
66
67 /*
68 ** Create semaphore for timer
69 ** 1 = init_value
70 ** 0 = priority order
71 */
72 timer_semaphore = sc_create(1, 0, &err);
73 if (err) output_routine("Error creating timer semaphore");
74
75 /*
76 ** The malloc semaphore controls access to the malloc and free
77 ** functions. This avoids re-entrancy problems associated with
78 ** a multi-tasking environment.
79 */
80 malloc_semaphore = sc_create(1, 0, &err);
81 if (err) output_routine("Error creating malloc semaphore");
82
83 /*
84 ** end of play interrupt semaphore
85 */
86 play_semaphore = sc_create(0, 1, &err);
87 if (err) sys_out("Error creating event flag for EOP interrupt.");
88
89 /*
90 ** create semaphore for lance rcv ISR
91 ** 0 = init_value
92 ** 0 = priority order
93 */
94 rcv_lance_pkt_semaphore = sc_create(0, 0, &err);
95 if (err) output_routine("Error creating rcv lance pkt semaphore");
96
97 if (get_lance_ready_to_go() != SUCCESS) {
98     output_routine("Error getting the LANCE ready to go.");
99     sc_tasuspend(0, 0, &err);
100 }
101
102 if (start_lance() != SUCCESS) {
103     output_routine("Error starting the LANCE.");
104     sc_tasuspend(0, 0, &err);
105 }
106
107 /*
108 ** spin of the tasks
109 */
110 sc_tcreate(housekeeping_task, HOUSEKEEPING_TASK_ID,
111 &err);
112 if (err) output_routine("Error creating housekeeping task.");
113
114 sc_tcreate(tx_task, TRANSMIT_TASK_ID, TRANSMIT_TASK_PRI, &err);
115 if (err) output_routine("Error creating transmit task.");
116
117 sc_tcreate(receive_task, RECEIVE_TASK_ID, RECEIVE_TASK_PRI, &err);
118 if (err) output_routine("Error creating receive task.");
119
120 sc_tcreate(pattern_task, PATTERN_TASK_ID, PATTERN_TASK_PRI, &err);
121 if (err) output_routine("Error creating pattern task.");

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hex.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:41 pm | 2/58 |

| LINE # | SOURCE TEXT |
|--------|--|
| 121 | |
| 122 | /* End of main task, main task commits suicide*/ |
| 123 | sc_delete(0, 0, task); |
| 124 | |
| 125 | /* end main */ |
| 126 | |
| 127 | |
| 128 | /*.....*/ |
| 129 | /* sys_out */ |
| 130 | /* Sends message to serial Channel A */ |
| 131 | /*.....*/ |
| 132 | |
| 133 | sys_out(buf) |
| 134 | { |
| 135 | char *buf, |
| 136 | while(*buf) |
| 137 | { |
| 138 | sc_putc(*buf++); |
| 139 | } |
| 140 | /* end sys_out */ |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hwconfig.c

DATE 5/23/89
TIME 6:14:42 pm

PAGE #
1/59

```

1  /* SCCS ID: hwconfig.c rev 1.1, 4/24/89 at 07:53:08 */
2
3  #include "device.h"
4  #include "message.h"
5  #include "hardware.h"
6  #include "mod_err.h"
7  #include "eeprom.h"
8  #include "laser.h"
9  #include "network.h"
10 #include "cpu.h"
11
12 #ifdef MODELER
13 #include "vtx.h"
14 #endif
15
16 extern LM_HARDWARE_ERROR modeler_error;
17 extern CONFIGURATION_ERRORS config_error;
18
19 #define DAB_INFO *bad_dab_list(MAX_LANE_COUNT * MAX_SLOT_COUNT);
20
21 #ifdef DEBASE
22 /* Set this variable to FALSE (using adb) to skip TNG calibration. */
23 #define do_tng_calibration TRUE
24 #else
25 #define do_tng_calibration FALSE
26 #endif
27
28 #define LIGHT_FAULT_LED() \
29     (((cpu_control_reg_struct *)CPU_CONTROL_REG)->fault_led = LED_ON)
30
31 read_hw_config()
32 {
33     /* Read the hardware configuration and fill in the data structure */
34
35     SYSTEM_INFO *new_system_info();
36     LANE_INFO *new_lane_info();
37     PAM_INFO *new_pam_info();
38     PEL_INFO *new_pel_info();
39     LANE_INFO *lane_ptr;
40     PEL_INFO *pel_ptr;
41     u_long board_id;
42     u_char lanemo, pammo, slotno;
43     u_char fb_block_index;
44
45 #ifdef DIRECTCONN
46     dc_init();
47 #endif
48
49     init_dab_eeprom_info();
50
51     system_config = new_system_info();
52
53     (void)read_cpu(&system_config->tot_system_mem,
54                  &system_config->slot_count,
55                  &system_config->phy_lane_count);
56
57     if (read_tng(&system_config->clab_board_id) == FAILURE) {
58         LIGHT_FAULT_LED();
59         lm_config_error(CERR_NO_TNG, 0, 0);
60         write_config_error();
61         return;
62     }
63
64 #ifdef DIRECTCONN
65     if (dc_init_pac() == FAILURE) {
66         DPRINTF(("Error from dc_init_pac\n"));
67         exit(1);
68     }
69 #endif
70
71 #ifdef MODELER
72     if (backplane_reset() == FAILURE)
73         DPRINTF(("ERROR in backplane_reset()\n"));
74 #endif
75
76 /* DEBASE: don't do anything */
77 #endif
78
79 if (do_tng_calibration == TRUE) {
80     DPRINTF(("calling tng_calibrate()\n"));
81     if (tng_calibrate() == FAILURE) {
82         DPRINTF(("FAILURE in tng_calibrate()\n"));
83         LIGHT_FAULT_LED();
84         lm_config_error(CERR_TNG_CAL, 0, 0);
85         write_config_error();
86         return;
87     }
88 }
89
90 for (lanemo = 0; lanemo < MAX_LANE_COUNT; lanemo++) {
91     /* check if the lane exists */
92     lane_ptr = new_lane_info();
93     system_config->lane[lanemo] = lane_ptr;
94
95     if (read_pac(lanemo) == SUCCESS) {
96         DPRINTF(("found lane: %d\n", lanemo));
97         lane_ptr->pac_present = TRUE;
98
99         (void)configure_pam(lanemo, lane_ptr);
100
101         fb_block_index = 0;
102         for (pammo = 0; pammo < MAX_PAM_COUNT; pammo++) {
103             if (lane_ptr->pam[pammo] != NULL) {
104                 if (fb_block_index == 0)
105                     fb_block_count_array[lanemo].PAM_max_addr[fb_block_index] =
106                         lane_ptr->pam[pammo] - lane_ptr->mem_size + PTRN_ADDR_INC;
107                 else
108                     fb_block_count_array[lanemo].PAM_max_addr[fb_block_index] =
109                         fb_block_count_array[lanemo].PAM_max_addr[fb_block_index - 1] +
110                         lane_ptr->pam[pammo] - lane_ptr->mem_size + PTRN_ADDR_INC;
111                 fb_block_count_array[lanemo].
112                     feedback_block_count[fb_block_index++] =
113                         fb_block_count_for_PAM(lane_ptr->pam[pammo] - lane_ptr->mem_size);
114             }
115         }
116     }
117 }
118
119 }
120

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hwconfig.c

*

DATE 5/23/89
TIME 6:14:42 pm

PAGE #
2/60

```

121  }
122  }
123
124  /* find all of the PEL's in this lane */
125  for (slotno = 0; slotno < MAX_SLOT_COUNT; slotno++) {
126      if (read_pel(lane--, slotno, &board_id)) {
127          pel_ptr = new_pel_info();
128          lane_ptr->pel[slotno] = pel_ptr;
129          pel_ptr->board_id = board_id;
130      }
131  }
132  }
133
134  init_bad_dab_list();
135
136  read_dab_configuration(dab_list);
137
138  verify_dab_list(dab_list);
139
140  /* We have to reset MAGIC ERRORS for ALL magic chips in the system; the
141   * good ones and the bad ones.
142   */
143
144  reset_magic_error(dab_list);
145  reset_magic_error(bad_dab_list);
146
147  raise_pel_reset();
148
149  xls_bad_dab_list();
150
151  measure_out_vcc(dab_list);
152
153  build_free_block_list();
154
155  write_config_error();
156
157  }
158
159  build_free_block_list()
160  {
161      LANE_INFO *lane_ptr;
162      u_long total_mem;
163      u_long link_table_addr;
164      u_long max_link_table_addr;
165      u_long block_addr;
166      u_short i;
167      u_short block_count;
168      u_char lane_no;
169      u_char panno;
170
171      for (lane_no = 0; lane_no < MAX_LANE_COUNT; ++lane_no) {
172          lane_ptr = system_config->lane[lane_no];
173
174          if (lane_ptr->pac_present == FALSE) {
175              free_block_list[lane_no] = NULL;
176              continue;
177          }
178
179          /* Calculate total pattern memory */
180          total_mem = 0;
181          for (panno = 0; panno < MAX_PAN_COUNT; ++panno) {
182              if (lane_ptr->pan[panno] != NULL) {
183                  total_mem += lane_ptr->pan[panno]->mem_size;
184              }
185          }
186
187          if (total_mem) {
188              /* make the free_block_list point to block number 1 (second block) */
189              free_block_list[lane_no] = lane_offset(lane_no) +
190                  BLOCK_ADDR_INC;
191
192              /* Mark all blocks as being free in the link table */
193              link_table_addr = lane_offset(lane_no) + LANE_LINK_TABLE_OFFSET;
194
195              block_count = total_mem / PTEN_PER_BLOCK;
196              max_link_table_addr = link_table_addr +
197                  block_count * LINK_TABLE_ADDR_INC;
198
199              while (link_table_addr != max_link_table_addr) {
200                  write_loc_long((u_long *)link_table_addr, (u_long)FREE_BLOCK_FLAG);
201                  link_table_addr += LINK_TABLE_ADDR_INC;
202              }
203
204              /* link all of the blocks */
205              block_addr = free_block_list[lane_no];
206              for (i = 1; i < block_count; ++i) {
207                  /* write forward link */
208                  write_loc_long((u_long *)block_addr, block_addr + BLOCK_ADDR_INC);
209
210                  /* write backward link */
211                  write_loc_long((u_long *)block_addr + 4,
212                      block_addr - BLOCK_ADDR_INC);
213
214                  block_addr += BLOCK_ADDR_INC;
215              }
216
217              /* Fix the backward pointer of the HEAD of the list */
218              write_loc_long((u_long *)free_block_list[lane_no] + 4, (u_long)NULL);
219
220              /* Fix the forward pointer of the TAIL of the list */
221              write_loc_long((u_long *)block_addr - BLOCK_ADDR_INC, (u_long)NULL);
222          }
223          else
224              free_block_list[lane_no] = NULL;
225      }
226  }
227
228  long calculate_pel_count(lane_number)
229  u_char lane_number;
230  {
231      DAB_INFO *temp;
232      u_char index;
233      char i;
234      u_char total_pel = 0;
235      u_char dabno;
236
237      calculate_pel_count_in_segment();
238  }

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/hwconfig.c | DATE 5/23/89 | PAGE # 3/61 |
|--|---|-------------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 241 | for (dabno = 0; dabno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++dabno) { | | | |
| 242 | if (dab_list[dabno] == NULL) | | | |
| 243 | continue; | | | |
| 244 | temp = dab_list[dabno]; | | | |
| 245 | if (temp->segment[0] != NULL) | | | |
| 246 | total_pel += calculate_pel_count_in_segment(lane_number, | | | |
| 247 | temp->segment[0]); | | | |
| 248 | else { | | | |
| 249 | index = 1; | | | |
| 250 | for (i = temp->segment_count; i > 0; --i) { | | | |
| 251 | total_pel += calculate_pel_count_in_segment(lane_number, | | | |
| 252 | temp->segment[index++]); | | | |
| 253 | } | | | |
| 254 | } | | | |
| 255 | return(total_pel); | | | |
| 256 | } | | | |
| 257 | u_char calculate_pel_count_in_segment(lane_number, segment) | | | |
| 258 | u_char lane_number; | | | |
| 259 | SEGMENT_EL *segment; | | | |
| 260 | { | | | |
| 261 | u_long mask; | | | |
| 262 | u_long shape_bit_map; | | | |
| 263 | u_char i; | | | |
| 264 | u_char count = 0; | | | |
| 265 | if (segment->lane_no > lane_number) | | | |
| 266 | return(0); | | | |
| 267 | else { | | | |
| 268 | shape_bit_map = segment->shape_bit_map; | | | |
| 269 | mask = 1 << (lane_number - segment->lane_no) * 8; | | | |
| 270 | for (i = 0; i < 8; ++i, mask <<= 1) { | | | |
| 271 | if (shape_bit_map & mask) | | | |
| 272 | ++count; | | | |
| 273 | } | | | |
| 274 | return(count); | | | |
| 275 | } | | | |
| 276 | dab_loc(dab_ptr, dab_no) | | | |
| 277 | DAB_INFO *dab_ptr; | | | |
| 278 | u_long dab_no; | | | |
| 279 | { | | | |
| 280 | u_long pro_number_of_locations_mark; | | | |
| 281 | long *pro_lane_slot_array; | | | |
| 282 | u_long shape_bit_map; | | | |
| 283 | mask; | | | |
| 284 | u_char lane_no, slotno; | | | |
| 285 | u_char bit_no; | | | |
| 286 | u_char seg_no; | | | |
| 287 | u_char i, j; | | | |
| 288 | u_char temp_lane; | | | |
| 289 | u_char temp_slot; | | | |
| 290 | u_char loc_count; | | | |
| 291 | pro_number_of_locations_mark = LM_MARK_BUFFER(lm_global_cons_ptr, | | | |
| 292 | lm_put_int(0); /* advance the obuf ptr */ | | | |
| 293 | pro_lane_slot_array = (long *)lm_get_cur_obuf_addr(); | | | |
| 294 | loc_count = 0; | | | |
| 295 | if (dab_ptr->segment[0] != NULL) { | | | |
| 296 | lane_no = dab_ptr->segment[0]->lane_no; | | | |
| 297 | slotno = dab_ptr->segment[0]->slotno; | | | |
| 298 | shape_bit_map = dab_ptr->segment[0]->shape_bit_map; | | | |
| 299 | for (mask = 1, bit_no = 0; bit_no < 32; mask <<= 1, ++bit_no) { | | | |
| 300 | if (mask & shape_bit_map) { | | | |
| 301 | lm_put_int((long)(lane_no + bit_no / 8)); | | | |
| 302 | lm_put_int((long)(slotno + bit_no % 8)); | | | |
| 303 | ++loc_count; | | | |
| 304 | } | | | |
| 305 | } | | | |
| 306 | else { | | | |
| 307 | seg_no = dab_no + 1; | | | |
| 308 | lane_no = dab_ptr->segment[seg_no]->lane_no; | | | |
| 309 | slotno = dab_ptr->segment[seg_no]->slotno; | | | |
| 310 | shape_bit_map = dab_ptr->segment[seg_no]->shape_bit_map; | | | |
| 311 | for (mask = 1, bit_no = 0; bit_no < 32; mask <<= 1, ++bit_no) { | | | |
| 312 | if (mask & shape_bit_map) { | | | |
| 313 | lm_put_int((long)(lane_no + bit_no / 8)); | | | |
| 314 | lm_put_int((long)(slotno + bit_no % 8)); | | | |
| 315 | ++loc_count; | | | |
| 316 | } | | | |
| 317 | } | | | |
| 318 | LM_PUT_LONG_AT_MARK(pro_number_of_locations_mark, lm_global_cons_ptr, | | | |
| 319 | loc_count); | | | |
| 320 | /* Now sort the array according to the lane number */ | | | |
| 321 | for (i = 0; i < loc_count - 1; ++i) { | | | |
| 322 | for (j = i + 1; j < loc_count; ++j) { | | | |
| 323 | if (pro_lane_slot_array[i*2] > pro_lane_slot_array[j*2]) { | | | |
| 324 | temp_lane = pro_lane_slot_array[i*2]; | | | |
| 325 | temp_slot = pro_lane_slot_array[i*2+1]; | | | |
| 326 | pro_lane_slot_array[i*2] = pro_lane_slot_array[j*2]; | | | |
| 327 | pro_lane_slot_array[i*2+1] = pro_lane_slot_array[j*2+1]; | | | |
| 328 | pro_lane_slot_array[j*2] = temp_lane; | | | |
| 329 | pro_lane_slot_array[j*2+1] = temp_slot; | | | |
| 330 | } | | | |
| 331 | } | | | |
| 332 | /* Then sort the array according to the slot number */ | | | |
| 333 | for (i = 0; i < loc_count - 1; ++i) { | | | |
| 334 | for (j = i + 1; j < loc_count; ++j) { | | | |
| 335 | if (pro_lane_slot_array[i*2] == pro_lane_slot_array[j*2]) { | | | |
| 336 | if (pro_lane_slot_array[i*2+1] > pro_lane_slot_array[j*2+1]) { | | | |
| 337 | temp_lane = pro_lane_slot_array[i*2]; | | | |
| 338 | temp_slot = pro_lane_slot_array[i*2+1]; | | | |
| 339 | pro_lane_slot_array[i*2] = pro_lane_slot_array[j*2]; | | | |
| 340 | pro_lane_slot_array[i*2+1] = pro_lane_slot_array[j*2+1]; | | | |
| 341 | pro_lane_slot_array[j*2] = temp_lane; | | | |
| 342 | pro_lane_slot_array[j*2+1] = temp_slot; | | | |
| 343 | } | | | |
| 344 | } | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hwconfig.c

DATE 5/23/89 PAGE #
TIME 6:14:42 pm 4/62

```

LINE # SOURCE TEXT
361     pro_lane_slot_array[i*2] = pro_lane_slot_array[j*2],
362     pro_lane_slot_array[i*2+1] = pro_lane_slot_array[j*2+1],
363
364     pro_lane_slot_array[j*2] = temp_lane;
365     pro_lane_slot_array[j*2+1] = temp_slot;
366
367 }
368
369 }
370
371
372 read_dab_configuration(loc_dab_list)
373 DAB_INFO *loc_dab_list[];
374 {
375     DAB_INFO *dab_ptr;
376     DAB_INFO *dab_ptr2;
377     SEGMENT_EL *seg;
378     DAB_INFO *new_dab_info();
379     SEGMENT_EL *new_segment();
380     u_long mask;
381     u_long dab_shape;
382     u_char ptr, *end;
383     u_short insertion_count;
384     u_char count;
385     dab_flag[MAX_LANE_COUNT][MAX_SLOT_COUNT];
386     char revision[REV_LENGTH + 1];
387     char dab_type[TYPE_LENGTH + 1];
388     char man_id[MAN_LENGTH + 1];
389     char model_maker[MAKER_LENGTH + 1];
390     char model_revision[REV_LENGTH + 1];
391     u_char slotno;
392     u_char laneso;
393     char dab_info_index;
394     u_char dab_segment_sum;
395
396     ptr = (u_char *)dab_flag[0][0];
397     end = &dab_flag[MAX_LANE_COUNT][0]; /* points to loc just past the array */
398     while (ptr < end)
399         *ptr++ = 0;
400
401     for (slotno = 0; slotno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++slotno)
402         loc_dab_list[slotno] = NULL;
403
404     for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {
405         /* Find all of the DAB's in this lane */
406         dab_ptr = new_dab_info();
407         for (slotno = 0; slotno < MAX_SLOT_COUNT; slotno++) {
408             if (!dab_flag[laneso][slotno]) {
409                 dab_flag[laneso][slotno] = 1;
410
411                 if (read_dab(laneso, slotno,
412                     man_id,
413                     dab_type,
414                     revision,
415                     dab_ptr->part_name,
416                     model_maker,
417                     model_revision,
418                     &dab_shape,
419                     &dab_segment_sum,
420                     &insertion_count) == SUCCESS) {
421
422                     for (count = 0; mask = 1; count < 32; mask <= 1, ++count) {
423                         if (dab_shape & mask)
424                             dab_flag[laneso + count / 8][slotno + count % 8] = 1;
425                     }
426
427                     if ((short)(dab_info_index = find_dab(loc_dab_list,
428                         dab_ptr->part_name,
429                         (u_char)PUBLIC)) != -1)
430                         dab_ptr2 = loc_dab_list[dab_info_index];
431                     else
432                         dab_ptr2 = NULL;
433
434                     if (dab_ptr2 == NULL) {
435                         enter_dab_info(loc_dab_list, dab_ptr, laneso, slotno);
436
437                         seg = new_segment();
438
439                         dab_ptr->segment[dab_segment_sum] = seg;
440
441                         seg->shape_bit_map = dab_shape;
442                         seg->lane_no = laneso;
443                         seg->slot_no = slotno;
444                         seg->insertion_count = insertion_count;
445                         (void)strcpy(seg->revision, revision);
446                         (void)strcpy(seg->dab_type, dab_type);
447                         (void)strcpy(seg->man_id, man_id);
448                         (void)strcpy(seg->model_maker, model_maker);
449                         (void)strcpy(seg->model_revision, model_revision);
450
451                         dab_ptr = new_dab_info();
452
453                     }
454                     else {
455                         if (dab_segment_sum != 0) {
456                             if (dab_ptr2->segment[dab_segment_sum] != NULL) {
457                                 /* The same kind of DAB has been seen before */
458                                 DPRINTF(("error: duplicate segment\n"));
459                                 lm_config_error(CERR_DUPLICATE_SEGMENT,
460                                     laneso, slotno);
461                             }
462                             else {
463                                 /* Other segment of the same part has been seen
464                                  * before.
465                                  */
466
467                                 seg = new_segment();
468
469                                 dab_ptr2->segment[dab_segment_sum] = seg;
470
471                                 seg->shape_bit_map = dab_shape;
472                                 seg->lane_no = laneso;
473                                 seg->slot_no = slotno;
474                                 seg->insertion_count = insertion_count;
475                                 (void)strcpy(seg->revision, revision);
476                                 (void)strcpy(seg->dab_type, dab_type);
477                                 (void)strcpy(seg->man_id, man_id);
478                                 (void)strcpy(seg->model_maker, model_maker);
479                                 (void)strcpy(seg->model_revision, model_revision);
480

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/hwconfig.c | DATE 5/23/89 TIME 6:14:42 pm | PAGE # 5/63 |
|--|--|-------------------------------------|---------------------------------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 481 | 1 | | | |
| 482 | 1 | | | |
| 483 | else { | | | |
| 484 | enter_dab_info(loc_dab_list, dab_ptr, laneso, slotno); | | | |
| 485 | seg = new_segment(); | | | |
| 486 | dab_ptr->segment[0] = seg; | | | |
| 487 | seg->shape_bit_map = dab_shape; | | | |
| 488 | seg->lane_no = laneso; | | | |
| 489 | seg->slot_no = slotno; | | | |
| 490 | seg->insertion_count = insertion_count; | | | |
| 491 | (void)strcpy(seg->revision, revision); | | | |
| 492 | (void)strcpy(seg->dab_type, dab_type); | | | |
| 493 | (void)strcpy(seg->mac_id, mac_id); | | | |
| 494 | (void)strcpy(seg->model_maker, model_maker); | | | |
| 495 | (void)strcpy(seg->model_revision, model_revision); | | | |
| 496 | dab_ptr = new_dab_info(); | | | |
| 497 | 1 | | | |
| 498 | 1 | | | |
| 499 | 1 | | | |
| 500 | 1 | | | |
| 501 | 1 | | | |
| 502 | 1 | | | |
| 503 | 1 | | | |
| 504 | 1 | | | |
| 505 | 1 | | | |
| 506 | 1 | | | |
| 507 | 1 | | | |
| 508 | 1 | | | |
| 509 | 1 | | | |
| 510 | 1 | | | |
| 511 | reset_magic_error(loc_dab_list) | | | |
| 512 | DAB_INFO *loc_dab_list(); | | | |
| 513 | { | | | |
| 514 | DAB_INFO *dab_ptr; | | | |
| 515 | u_long dab_shape; | | | |
| 516 | u_long junk; | | | |
| 517 | u_long mask; | | | |
| 518 | u_long magic_addr; | | | |
| 519 | u_char lanesobase; | | | |
| 520 | u_char slotno; | | | |
| 521 | u_char laneso; | | | |
| 522 | u_char slotno; | | | |
| 523 | u_char dabno; | | | |
| 524 | u_char count; | | | |
| 525 | u_char i; | | | |
| 526 | /* Reset each MAGIC chip error register in the system and also set the | | | |
| 527 | * ACTIVE bit. | | | |
| 528 | /* Note that at this point the loc_dab_list has not been verified yet, | | | |
| 529 | * so there could be missing segments. | | | |
| 530 | */ | | | |
| 531 | for (dabno = 0; dabno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++dabno) { | | | |
| 532 | if (loc_dab_list[dabno] == NULL) | | | |
| 533 | continue; | | | |
| 534 | dab_ptr = loc_dab_list[dabno]; | | | |
| 535 | /* We don't need to reset magic if all units of the DABs are active */ | | | |
| 536 | if (all_dab_unit_active_and_initialized(dab_ptr) == TRUE) { | | | |
| 537 | DPRINTF(("dab: %d active --> don't reset", dabno)); | | | |
| 538 | continue; | | | |
| 539 | DPRINTF(("reset dab: %d", dabno)); | | | |
| 540 | if (dab_ptr->segment[0] != NULL) { | | | |
| 541 | /* This is a single segment device */ | | | |
| 542 | lanesobase = dab_ptr->segment[0]->lane_no; | | | |
| 543 | slotno = dab_ptr->segment[0]->slot_no; | | | |
| 544 | dab_shape = dab_ptr->segment[0]->shape_bit_map; | | | |
| 545 | for (count = 0, mask = 1; count < 32; mask <= 1, ++count) { | | | |
| 546 | if (dab_shape & mask) { | | | |
| 547 | lanesno = lanesobase + count / 8; | | | |
| 548 | slotno = slotno + count % 8; | | | |
| 549 | if (read_pel(lanesno, slotno, &junk) == SUCCESS) { | | | |
| 550 | /* We only need to read reg 15 from ONE of the MAGIC Chip | | | |
| 551 | * to reset the error register of ALL MAGIC Chips. | | | |
| 552 | */ | | | |
| 553 | magic_addr = LANE_0_START_ADDR + lanesno * LANE_ADDR_INC + | | | |
| 554 | LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC + | | | |
| 555 | PEL_MC_0_OFFSET; | | | |
| 556 | (void)read_loc_long((u_long *) | | | |
| 557 | (magic_addr + MC_RESET_REG_OFFSET)); | | | |
| 558 | /* Need to read some other Magic register to affect reset */ | | | |
| 559 | (void)read_loc_long((u_long *) | | | |
| 560 | (magic_addr + MC_DATA_OUT_REG_OFFSET)); | | | |
| 561 | init_dab(lanesno, slotno); | | | |
| 562 | } | | | |
| 563 | } | | | |
| 564 | else { | | | |
| 565 | /* This is a multi segments device */ | | | |
| 566 | for (i = 0; i < MAX_SEGMENT_PER_DEVICE; ++i) { | | | |
| 567 | if (dab_ptr->segment[i] == NULL) | | | |
| 568 | continue; | | | |
| 569 | lanesobase = dab_ptr->segment[i]->lane_no; | | | |
| 570 | slotno = dab_ptr->segment[i]->slot_no; | | | |
| 571 | dab_shape = dab_ptr->segment[i]->shape_bit_map; | | | |
| 572 | for (count = 0, mask = 1; count < 32; mask <= 1, ++count) { | | | |
| 573 | if (dab_shape & mask) { | | | |
| 574 | lanesno = lanesobase + count / 8; | | | |
| 575 | slotno = slotno + count % 8; | | | |
| 576 | if (read_pel(lanesno, slotno, &junk) == SUCCESS) { | | | |
| 577 | /* We only need to read reg 15 from ONE of the MAGIC Chip | | | |
| 578 | * to reset the error register of ALL MAGIC Chips. | | | |
| 579 | */ | | | |
| 580 | magic_addr = LANE_0_START_ADDR + lanesno * LANE_ADDR_INC + | | | |
| 581 | LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC + | | | |
| 582 | PEL_MC_0_OFFSET; | | | |
| 583 | (void)read_loc_long((u_long *) | | | |
| 584 | (magic_addr + MC_RESET_REG_OFFSET)); | | | |
| 585 | /* Need to read some other Magic register to affect reset */ | | | |
| 586 | (void)read_loc_long((u_long *) | | | |
| 587 | (magic_addr + MC_DATA_OUT_REG_OFFSET)); | | | |
| 588 | init_dab(lanesno, slotno); | | | |
| 589 | } | | | |
| 590 | } | | | |
| 591 | } | | | |
| 592 | } | | | |
| 593 | } | | | |
| 594 | } | | | |
| 595 | } | | | |
| 596 | } | | | |
| 597 | } | | | |
| 598 | } | | | |
| 599 | } | | | |
| 600 | } | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hwconfig.c

DATE 5/23/89 PAGE #
TIME 6:14:42 pm 6/64

```

LINE # SOURCE TEXT
601
602 (void)read_loc_long((u_long *)
603 (magic_addr + MC_RESET_REG_OFFSET));
604
605 /* Need to read some other Magic reg to affect reset */
606 (void)read_loc_long((u_long *)
607 (magic_addr + MC_DATA_OUT_REG_OFFSET));
608
609 (void)read_loc_long((u_long *)magic_addr);
610
611 init_dab(laneno, slotno);
612
613 }
614
615 }
616
617 }
618
619
620 verify_dab_list(loc_dab_list);
621 DAB_INFO *loc_dab_list[];
622 {
623 /* Verify that
624 * - all of the PEL's exist to support the DAB's.
625 * - the PMA's exist in each lane occupied by the DAB
626 * - all segments of parts exists.
627 * - Duplicate devices have the same unit_location
628 * Set up the unit_addr field in the dab_info.
629 * Set up the last_in_lane field of the unit_addr_el
630 * Set up lane_used[]
631 * Set up lane_count
632 * Set up unit_count_per_lane
633 * Set up the dummy_ptr
634 * Set up ident_lane
635 * Make the lane point to the most significant DAB.
636 */
637
638 DAB_INFO *dab_ptr;
639 DAB_INFO *dab_ptr2;
640 u_long mask;
641 u_long dab_shape;
642 u_char pel_location[MAX_LANE_COUNT][MAX_SLOT_COUNT];
643 u_char cur_unit;
644 u_char laneno;
645 u_char slotno;
646 u_char count;
647 u_char i, j;
648 u_char last_segment_sum;
649 u_char dabno;
650 u_char dabno2;
651
652 for (dabno = 0; dabno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++dabno) {
653
654 if (loc_dab_list[dabno] == NULL)
655 continue;
656
657 dab_ptr = loc_dab_list[dabno];
658
659 if (dab_ptr->segment[0] != NULL) {
660 /* This is a single segment device */
661
662 dab_ptr->segment_count = 1;
663 cur_unit = 0;
664 laneno = dab_ptr->segment[0]->lane_no;
665 slotno = dab_ptr->segment[0]->slot_no;
666 dab_shape = dab_ptr->segment[0]->shape_bit_map;
667
668 for (count = 0, mask = 1; count < 32; mask <= 1, ++count) {
669 if (dab_shape & mask) {
670 dab_ptr->unit_location[cur_unit].lane_no = laneno + count / 8;
671 dab_ptr->unit_location[cur_unit].slot_no = slotno + count % 8;
672 ++cur_unit;
673 }
674 }
675
676 dab_ptr->unit_count = cur_unit;
677
678 if (check_pel_and_pam(dab_ptr) == FAILURE)
679 remove_dab_info(loc_dab_list, dabno);
680 else
681 set_last_in_lane(dab_ptr);
682 }
683 else {
684 /* This is a multi segments device */
685 /* Find the last segment */
686 for (i = MAX_SEGMENT_PER_DEVICE, i > 0, --i) {
687 if (dab_ptr->segment[i] != NULL)
688 break;
689 }
690
691 dab_ptr->segment_count = i;
692
693 cur_unit = 0;
694 last_segment_sum = 1;
695 for (i = 1; i <= last_segment_sum; i++) {
696 if (dab_ptr->segment[i] == NULL)
697 break;
698 else {
699 laneno = dab_ptr->segment[i]->lane_no;
700 slotno = dab_ptr->segment[i]->slot_no;
701 dab_shape = dab_ptr->segment[i]->shape_bit_map;
702
703 for (count = 0, mask = 1; count < 32; mask <= 1, ++count) {
704 if (dab_shape & mask) {
705 if (cur_unit == MAX_UNIT_COUNT) {
706 for (j = 1; j <= last_segment_sum; j++) {
707 if (j != i) {
708 laneno = dab_ptr->segment[j]->lane_no;
709 slotno = dab_ptr->segment[j]->slot_no;
710 }
711 }
712
713 remove_dab_info(loc_dab_list, dabno);
714
715 goto next;
716 }
717
718 dab_ptr->unit_location[cur_unit].lane_no =
719 laneno + count / 8;
720 dab_ptr->unit_location[cur_unit].slot_no =
721 slotno + count % 8;

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/hwconfig.c | DATE 5/23/89 | PAGE # 7/65 |
|--|---|-------------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 722 | ++cur_unit; | | | |
| 723 | } | | | |
| 724 | } | | | |
| 725 | } | | | |
| 726 | } | | | |
| 727 | dab_ptr->unit_count = cur_unit; | | | |
| 728 | if (i != last_segment_num) { | | | |
| 729 | /* Not all of the segments of the device were found. | | | |
| 730 | * Consider this device unusable and remove it from the dab_list | | | |
| 731 | */ | | | |
| 732 | for (i = 1; i <= MAX_SEGMENT_PER_DEVICE; ++i) { | | | |
| 733 | if (dab_ptr->segment[i] == NULL) | | | |
| 734 | continue; | | | |
| 735 | lm_config_error(CERR_MISSING_SEGMENT, | | | |
| 736 | dab_ptr->segment[i]->lane_no, | | | |
| 737 | dab_ptr->segment[i]->slot_no); | | | |
| 738 | } | | | |
| 739 | } | | | |
| 740 | remove_dab_info(loc_dab_list, dabno); | | | |
| 741 | } | | | |
| 742 | } | | | |
| 743 | /* All of the segments were found */ | | | |
| 744 | if (check_pel_and_pem(dab_ptr) == FAILURE) { | | | |
| 745 | remove_dab_info(loc_dab_list, dabno); | | | |
| 746 | } | | | |
| 747 | else { | | | |
| 748 | /* Make the lane point to the most significant segment */ | | | |
| 749 | set_last_lane(dab_ptr); | | | |
| 750 | } | | | |
| 751 | } | | | |
| 752 | } | | | |
| 753 | } | | | |
| 754 | } | | | |
| 755 | next; | | | |
| 756 | } | | | |
| 757 | } | | | |
| 758 | } | | | |
| 759 | /* Make sure that duplicate devices have the same relative unit location. | | | |
| 760 | * Note: only single segment devices can have duplicates at this point. | | | |
| 761 | */ | | | |
| 762 | for (dabno = 0; dabno < (MAX_LANE_COUNT * MAX_SLOT_COUNT) - 1; ++dabno) { | | | |
| 763 | if (loc_dab_list[dabno] == NULL) | | | |
| 764 | continue; | | | |
| 765 | dab_ptr = loc_dab_list[dabno]; | | | |
| 766 | for (dabno2 = dabno + 1; | | | |
| 767 | dabno2 < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++dabno2) { | | | |
| 768 | if (loc_dab_list[dabno2] == NULL) | | | |
| 769 | continue; | | | |
| 770 | dab_ptr2 = loc_dab_list[dabno2]; | | | |
| 771 | if (strcmp(dab_ptr->part_name, dab_ptr2->part_name) == 0) { | | | |
| 772 | if (cmp_dab_shape(dab_ptr, dab_ptr2) == FALSE) { | | | |
| 773 | lm_config_error(CERR_ILLEGAL_DUPLICATE_DEVICE, | | | |
| 774 | dab_ptr2->segment(0)->lane_no, | | | |
| 775 | dab_ptr2->segment(0)->slot_no); | | | |
| 776 | remove_dab_info(loc_dab_list, dabno2); | | | |
| 777 | } | | | |
| 778 | } | | | |
| 779 | /* Create a MAP of the PEL location to be used to check pel stacking */ | | | |
| 780 | for (laneo = 0; laneo < MAX_LANE_COUNT; ++laneo) { | | | |
| 781 | for (slotno = 0; slotno < MAX_SLOT_COUNT; ++slotno) { | | | |
| 782 | pel_location[laneo][slotno] = 0; | | | |
| 783 | if (system_config->lane[laneo]->pel[slotno] != NULL) | | | |
| 784 | pel_location[laneo][slotno] = 1; | | | |
| 785 | } | | | |
| 786 | } | | | |
| 787 | } | | | |
| 788 | for (dabno = 0; dabno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++dabno) { | | | |
| 789 | dab_ptr = loc_dab_list[dabno]; | | | |
| 790 | if (dab_ptr == NULL) | | | |
| 791 | continue; | | | |
| 792 | if (check_pel_stacking(dab_ptr, pel_location) == FAILURE) { | | | |
| 793 | if (dab_ptr->segment(0) != NULL) { | | | |
| 794 | /* This is a single segment device */ | | | |
| 795 | lm_config_error(CERR_ILLEGAL_PEL_STACKING, | | | |
| 796 | dab_ptr->segment(0)->lane_no, | | | |
| 797 | dab_ptr->segment(0)->slot_no); | | | |
| 798 | remove_dab_info(loc_dab_list, dabno); | | | |
| 799 | } | | | |
| 800 | else { | | | |
| 801 | /* This is a multi segment device */ | | | |
| 802 | for (i = 1; i <= MAX_SEGMENT_PER_DEVICE; ++i) { | | | |
| 803 | if (dab_ptr->segment[i] == NULL) | | | |
| 804 | continue; | | | |
| 805 | lm_config_error(CERR_ILLEGAL_PEL_STACKING, | | | |
| 806 | dab_ptr->segment[i]->lane_no, | | | |
| 807 | dab_ptr->segment[i]->slot_no); | | | |
| 808 | } | | | |
| 809 | remove_dab_info(loc_dab_list, dabno); | | | |
| 810 | } | | | |
| 811 | } | | | |
| 812 | } | | | |
| 813 | } | | | |
| 814 | } | | | |
| 815 | } | | | |
| 816 | } | | | |
| 817 | } | | | |
| 818 | } | | | |
| 819 | } | | | |
| 820 | } | | | |
| 821 | } | | | |
| 822 | } | | | |
| 823 | } | | | |
| 824 | } | | | |
| 825 | } | | | |
| 826 | } | | | |
| 827 | } | | | |
| 828 | } | | | |
| 829 | #ifdef DEBUG | | | |
| 830 | for (dabno = 0; dabno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++dabno) { | | | |
| 831 | if (loc_dab_list[dabno] == NULL) | | | |
| 832 | continue; | | | |
| 833 | dab_ptr = loc_dab_list[dabno]; | | | |
| 834 | DPRINTF(("device name : %s", dab_ptr->part_name)); | | | |
| 835 | DPRINTF(("unit_count_per_lane : %d", dab_ptr->unit_count_per_lane)); | | | |
| 836 | DPRINTF(("segment_count : %d", dab_ptr->segment_count)); | | | |
| 837 | DPRINTF(("lane_count : %d", dab_ptr->lane_count)); | | | |
| 838 | DPRINTF(("unit_count : %d", dab_ptr->unit_count)); | | | |
| 839 | } | | | |
| 840 | #endif | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hwconfig.c

DATE 5/23/89
TIME 6:14:42 pm

PAGE #
8/66

```

LINE #          SOURCE TEXT
841      DPRINTF(("unit location : %d\n"),
842      for (i = 0; i < dab_ptr->unit_count; ++i) {
843          DPRINTF((" lane: %d slot: %d\n",
844          dab_ptr->unit_location[i].lane_no,
845          dab_ptr->unit_location[i].slot_no));
846      }
847      }
848      }
849      }
850      }
851      check_pel_and_pam(dab_ptr)
852      DAB_INFO *dab_ptr;
853      }
854      /* Check to make sure that each unit of the device is supported by a PEL,
855      * also that it is backed by PAM.
856      */
857      }
858      short i;
859      u_char laneso, slotso;
860      u_char any_error = FALSE;
861      }
862      for (i = dab_ptr->unit_count - 1; i >= 0; --i) {
863          laneso = dab_ptr->unit_location[i].lane_no;
864          slotso = dab_ptr->unit_location[i].slot_no;
865          }
866          /* Check if the PEL exists on this slot */
867          if (system_config->lane(laneso)->pel(slotso) == NULL) {
868              lm_config_error(CERR_NO_PEL_FOR_DAB,
869              dab_ptr->unit_location[i].lane_no,
870              dab_ptr->unit_location[i].slot_no);
871              any_error = TRUE;
872          }
873          }
874          /* Check if the PAM exists */
875          if (system_config->lane(laneso)->pam_present == FALSE) {
876              lm_config_error(CERR_NO_PAM_FOR_DAB,
877              dab_ptr->unit_location[i].lane_no,
878              dab_ptr->unit_location[i].slot_no);
879              any_error = TRUE;
880              continue;
881          }
882          }
883          /* Check if the PAM exists on this lane */
884          if (system_config->lane(laneso)->pam[0] == NULL) {
885              lm_config_error(CERR_NO_PAM_FOR_DAB,
886              dab_ptr->unit_location[i].lane_no,
887              dab_ptr->unit_location[i].slot_no);
888              any_error = TRUE;
889          }
890          }
891          }
892          if (any_error == FALSE)
893              return(SUCCESS);
894          else
895              return(FAILURE);
896          }
897      }
898      remove_dab_info(loc_dab_list, dabso);
899      DAB_INFO *loc_dab_list[];
900      u_char dabso;
901      }
902      /* Remove the dab_info from the loc_dab_list and put it to bad_dab_list. */
903      DAB_INFO *dab_ptr;
904      }
905      dab_ptr = loc_dab_list[dabso];
906      }
907      bad_dab_list[dabso] = dab_ptr;
908      }
909      loc_dab_list[dabso] = NULL;
910      }
911      }
912      init_dab(laneso, slotso)
913      u_char laneso;
914      u_char slotso;
915      }
916      /* Initialize the DAB (set the ACTIVE bit) at the laneso/slotso. */
917      u_long addr;
918      u_short word;
919      }
920      addr = LANE_0_START_ADDR + laneso * LANE_ADDR_INC +
921      LANE_PEL_0_START_OFFSET + slotso * LANE_PEL_ADDR_INC +
922      PEL_STATUS_CONTROL_OFFSET;
923      }
924      /* assert RESET */
925      word = read_loc_long((u_long *)addr);
926      DPRINTF(("id: %d al: %d pel status: %04x\n", laneso, slotso, word));
927      word |= PEL_CS_RESET_MASK;
928      write_loc_long((u_long *)addr, (u_long)word);
929      }
930      /* write INITIALIZE to 0 with RESET == 0 */
931      word = read_loc_long((u_long *)addr);
932      word = (word & PEL_CS_INITIALIZE_MASK);
933      write_loc_long((u_long *)addr, (u_long)word);
934      }
935      /* write INITIALIZE to 1 with RESET == 0 */
936      word = read_loc_long((u_long *)addr);
937      word = word | PEL_CS_INITIALIZE_MASK;
938      write_loc_long((u_long *)addr, (u_long)word);
939      }
940      /* deassert RESET */
941      word = read_loc_long((u_long *)addr);
942      word |= PEL_CS_RESET_MASK;
943      }
944      /* ??? disable magic error */
945      word |= PEL_CS_MAGIC_ERROR_ENABLE_MASK;
946      }
947      write_loc_long((u_long *)addr, (u_long)word);
948      }
949      }
950      }
951      reconfigure_dab()
952      {
953          DAB_INFO *dev_dab_list[MAX_LANE_COUNT*MAX_SLOT_COUNT];
954          EXTRA_DEVICE_SPEC *extra_def_ptr;
955          USER *user;
956          DEVICE_SPEC *definition;
957          INSTANCE_INFO *instance;
958          DAB_INFO *dab_ptr;
959          DAB_INFO *new_dab_ptr;
960          u_short dab_info_index;

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hwconfig.c

DATE 5/23/89
TIME 6:14:42 pm

PAGE #
9/67

```

LINE # SOURCE TEXT
961 u_short inst_id;
962 u_char slotno;
963 u_char userno;
964 u_char again_count = 0;
965
966 again:
967 modeler_error.dab_change = FALSE;
968
969 DPRINTF(("waiting for DAB to settle\n"));
970 lm_delay((u_long)(1 - 1000));
971
972 if (modeler_error.error == TRUE) {
973     modeler_error.error = FALSE;
974
975     if ((modeler_error.pac_lane_errors != 0) ||
976         (modeler_error.tmg_error == TRUE) ||
977         (modeler_error.unknown_source_of_interrupt == TRUE)) {
978         fatal_hardware_error_encountered = TRUE;
979         return;
980     }
981 }
982
983 /* We are going to reconfigure the DABs --> reinitialize the structure */
984 config_error.duplicate_segment = 0;
985 config_error.missing_segment = 0;
986 config_error.no_pal_for_dab = 0;
987 config_error.no_pam_for_dab = 0;
988 config_error.illegal_duplicate_device = 0;
989 config_error.device_too_large = 0;
990 config_error.illegal_pal_stacking = 0;
991 non_fatal_configuration_error_encountered = FALSE;
992
993 init_bad_dab_list();
994
995 read_dab_configuration(new_dab_list);
996
997 update_svars_dab_insertion(new_dab_list);
998
999 verify_dab_list(new_dab_list);
1000
1001 for (userno = 0; userno < MAX_USER_COUNT; ++userno) {
1002     user = user_info_array[userno];
1003
1004     if (user->active == FALSE)
1005         continue;
1006
1007     for (inst_id = 0; inst_id < user->inst_table_size; ++inst_id) {
1008         instance = user->instance[inst_id];
1009
1010         if (BOCUS_INSTANCE(user, instance))
1011             continue;
1012
1013         definition = instance->definition;
1014
1015         extra_def_ptr = (EXTRA_DEVICE_SPEC *)definition->extra_data;
1016         if (extra_def_ptr->dab_ok == TRUE) {
1017             dab_info_index = instance->dab_info_index;
1018
1019             dab_ptr = dab_list[dab_info_index];
1020             new_dab_ptr = new_dab_list[dab_info_index];
1021
1022             if ((new_dab_ptr != NULL) &&
1023                 (strcmp(dab_ptr->part_name,
1024                     new_dab_ptr->part_name) == 0) &&
1025                 (strcmp(dab_ptr->location, new_dab_ptr->location) == TRUE)) {
1026                 /* The same DAB is present in the same location */
1027
1028                 if (all_dab_unit_active_and_initialized(dab_ptr) == FALSE) {
1029                     /* Check to see if the DAB is touched (removed and inserted
1030                      * back again) at all by looking at the ACTIVE bit.
1031                     */
1032                     if (definition->device_type == PRIVATE) {
1033                         /* If any of them is touched then invalidate PRIVATE
1034                          * device.
1035                         */
1036                         extra_def_ptr->dab_ok = FALSE;
1037                         instance->dab_info_index = -1;
1038                     }
1039                 }
1040             }
1041             else {
1042                 /* The DAB is removed or changed */
1043                 extra_def_ptr->dab_ok = FALSE;
1044                 instance->dab_info_index = -1;
1045             }
1046         }
1047
1048         /* Increment the DAB INFO.act_inst_count or DAB INFO.act_var_count,
1049          * set the DAB INFO.used_as_private appropriately.
1050         */
1051         if (extra_def_ptr->dab_ok == TRUE) {
1052             if (instance->is_fault == TRUE)
1053                 ++new_dab_ptr->act_var_count;
1054             else {
1055                 ++new_dab_ptr->act_inst_count;
1056                 if (definition->device_type == PRIVATE)
1057                     new_dab_ptr->used_as_private = TRUE;
1058             }
1059         }
1060     }
1061 }
1062
1063 /* We have to reset MAGIC ERRORS for ALL magic chips in the system; the
1064 * good ones and the bad ones.
1065 */
1066 reset_magic_error(new_dab_list);
1067 reset_magic_error(bad_dab_list);
1068
1069 raise_pal_reset();
1070
1071 rls_bad_dab_list();
1072
1073 measure_dut_vcc(new_dab_list);
1074
1075 /* release the old dab_list */
1076 for (slotno = 0; slotno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++slotno) {
1077     if (dab_list[slotno] != NULL) {
1078         rls_dab_info(dab_list[slotno]);
1079         dab_list[slotno] = NULL;
1080     }
1081 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hwconfig.c

DATE 5/23/89
TIME 6:14:42 pm

PAGE #
10/68

```

1081 }
1082 }
1083
1084 DPRINTF(("DAB found: \n"));
1085 /* copy the new dab into the dab_list */
1086 for (slotno = 0; slotno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++slotno) {
1087     if (new_dab_list[slotno] != NULL) {
1088         dab_list[slotno] = new_dab_list[slotno];
1089
1090         DPRINTF(("slotno", dab_list[slotno]->part_name));
1091
1092         /* Turn on IS USE led */
1093         if ((dab_list[slotno]->act_inst_count +
1094             dab_list[slotno]->act_var_count) != 0)
1095             turn_on_is_use(dab_list[slotno]);
1096     }
1097 }
1098
1099 clear_non_fatal_mod_err();
1100
1101 if (any_pel_with_reset_asserted() == TRUE) {
1102     if (again_count++ < 10) {
1103         DPRINTF(("some pels still have reset asserted\n"));
1104         goto again;
1105     }
1106     else {
1107         DPRINTF(("infinite loop detected in reconfigure_dab. Bailing out...\n"));
1108     }
1109 }
1110
1111 DPRINTF(("exiting reconfigure_dab\n"));
1112 }
1113
1114 any_pel_with_reset_asserted()
1115 {
1116     u_long   addr;
1117     u_short  word;
1118     u_char   lane_no;
1119     u_char   slotno;
1120
1121     for (lane_no = 0; lane_no < MAX_LANE_COUNT; ++lane_no) {
1122         for (slotno = 0; slotno < MAX_SLOT_COUNT; ++slotno) {
1123             if (system_config->lane[lane_no]->pel[slotno] == NULL)
1124                 continue;
1125
1126             addr = LANE_0_START_ADDR + lane_no * LANE_ADDR_INC +
1127                 LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC +
1128                 PEL_STATUS_CONTROL_OFFSET;
1129
1130             word = read_loc_low((u_long *)addr);
1131             DPRINTF(("apera: ln: %d sl: %d pel status: %04x\n",
1132                 lane_no, slotno, word));
1133
1134             if ((word & PEL_CS_RESET_MASK) == 0) {
1135                 DPRINTF(("pel reset on lane: %d slot: %d still asserted\n",
1136                     lane_no, slotno));
1137                 return(TRUE);
1138             }
1139         }
1140     }
1141     return(FALSE);
1142 }
1143
1144 cmp_dab_location(dab_ptr1, dab_ptr2)
1145 DAB_INFO *dab_ptr1;
1146 DAB_INFO *dab_ptr2;
1147 {
1148     /* compare the each unit position of dab_ptr1 and dab_ptr2.
1149      * Return TRUE if they are the same else return FALSE.
1150      */
1151
1152     u_char   unitno;
1153     u_char   total_unit;
1154
1155     if (dab_ptr1->unit_count != dab_ptr2->unit_count)
1156         return(FALSE);
1157
1158     if (dab_ptr1->unit_count_per_lane != dab_ptr2->unit_count_per_lane)
1159         return(FALSE);
1160
1161     total_unit = dab_ptr1->unit_count;
1162     for (unitno = 0; unitno < total_unit; ++unitno) {
1163         if ((dab_ptr1->unit_location[unitno].lane_no !=
1164             dab_ptr2->unit_location[unitno].lane_no) ||
1165             (dab_ptr1->unit_location[unitno].slot_no !=
1166             dab_ptr2->unit_location[unitno].slot_no))
1167             return(FALSE);
1168     }
1169     return(TRUE);
1170 }
1171
1172 cmp_dab_shape(dab_ptr1, dab_ptr2)
1173 DAB_INFO *dab_ptr1;
1174 DAB_INFO *dab_ptr2;
1175 {
1176     /* compare the each unit relative position of dab_ptr1 and dab_ptr2.
1177      * Return TRUE if they are the same else return FALSE.
1178      */
1179
1180     u_char   unitno;
1181     u_char   total_unit;
1182     u_char   lanesol;
1183     u_char   slotsol;
1184     u_char   lanesol2;
1185     u_char   slotsol2;
1186
1187     if (dab_ptr1->unit_count != dab_ptr2->unit_count)
1188         return(FALSE);
1189
1190     if (dab_ptr1->unit_count_per_lane != dab_ptr2->unit_count_per_lane)
1191         return(FALSE);
1192
1193     lanesol = dab_ptr1->unit_location[0].lane_no;
1194     lanesol2 = dab_ptr2->unit_location[0].lane_no;
1195     slotsol = dab_ptr1->unit_location[0].slot_no;
1196     slotsol2 = dab_ptr2->unit_location[0].slot_no;
1197
1198     total_unit = dab_ptr1->unit_count;
1199     for (unitno = 0; unitno < total_unit; ++unitno) {
1200

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/hwconfig.c | DATE 5/23/89 | PAGE # 11/69 |
|--|--|-------------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 1201 | if ((dab_ptr->unit_location(unitno).lane_no - lanesol != | | | |
| 1202 | dab_ptr->unit_location(unitno).lane_no - lanesol2) | | | |
| 1203 | (dab_ptr->unit_location(unitno).slot_no - slotsol != | | | |
| 1204 | dab_ptr->unit_location(unitno).slot_no - slotsol2)) | | | |
| 1205 | return("SE"); | | | |
| 1206 | } | | | |
| 1207 | } | | | |
| 1208 | return(TRUE); | | | |
| 1209 | } | | | |
| 1210 | measure_dut_vcc(loc_dab_list) | | | |
| 1211 | DAB_INFO *loc_dab_list[]; | | | |
| 1212 | { | | | |
| 1213 | DAB_INFO *dab_ptr; | | | |
| 1214 | u_long addr; | | | |
| 1215 | u_long delay; | | | |
| 1216 | u_char lanesol; | | | |
| 1217 | u_char slotsol; | | | |
| 1218 | u_char lanesol2; | | | |
| 1219 | u_char slotsol2; | | | |
| 1220 | u_char value; | | | |
| 1221 | } | | | |
| 1222 | for (dabno = 0; dabno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++dabno) { | | | |
| 1223 | { | | | |
| 1224 | if (loc_dab_list[dabno] == NULL) | | | |
| 1225 | continue; | | | |
| 1226 | dab_ptr = loc_dab_list[dabno]; | | | |
| 1227 | lanesol = dab_ptr->unit_location(0).lane_no; | | | |
| 1228 | slotsol = dab_ptr->unit_location(0).slot_no; | | | |
| 1229 | addr = LANE_0_START_ADDR + lanesol * LANE_ADDR_INC + | | | |
| 1230 | LANE_PEL_0_START_OFFSET + slotsol * LANE_PEL_ADDR_INC + | | | |
| 1231 | PEL_DOTVCC_DAC_READ_OFFSET; | | | |
| 1232 | /* The first measurement is always wrong after power up */ | | | |
| 1233 | #ifdef DEBUG | | | |
| 1234 | if (value = read_loc_long((u_long *)addr) & 0xff, | | | |
| 1235 | delay = 1000; | | | |
| 1236 | while (--delay) , | | | |
| 1237 | value = read_loc_long((u_long *)addr) & 0xff, | | | |
| 1238 | #else | | | |
| 1239 | value = 0x0; | | | |
| 1240 | #endif | | | |
| 1241 | dab_ptr->dut_vcc_measured = MAX_DOT_VCC_VOLTAGE * value / 256; | | | |
| 1242 | DPRINTF(("dut measured: %d\n", dab_ptr->dut_vcc_measured)); | | | |
| 1243 | } | | | |
| 1244 | } | | | |
| 1245 | init_bad_dab_list() | | | |
| 1246 | { | | | |
| 1247 | u_char slotno; | | | |
| 1248 | for (slotno = 0; slotno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++slotno) | | | |
| 1249 | bad_dab_list[slotno] = NULL; | | | |
| 1250 | } | | | |
| 1251 | rle_bad_dab_list() | | | |
| 1252 | { | | | |
| 1253 | u_char slotno; | | | |
| 1254 | for (slotno = 0; slotno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++slotno) { | | | |
| 1255 | if (bad_dab_list[slotno] == NULL) | | | |
| 1256 | continue; | | | |
| 1257 | rle_dab_info(bad_dab_list[slotno]); | | | |
| 1258 | bad_dab_list[slotno] = NULL; | | | |
| 1259 | } | | | |
| 1260 | all_dab_unit_active_and_initialized(dab_ptr) | | | |
| 1261 | DAB_INFO *dab_ptr; | | | |
| 1262 | { | | | |
| 1263 | /* Returns TRUE if all unit of DAB has the ACTIVE bit, the | | | |
| 1264 | INITIALIZE bit set, and RESET is deasserted. The INITIALIZE bit | | | |
| 1265 | is set to 0 when we are labeling the DAB. | | | |
| 1266 | */ | | | |
| 1267 | u_long addr; | | | |
| 1268 | u_short value; | | | |
| 1269 | u_char total_unit; | | | |
| 1270 | u_char unitno; | | | |
| 1271 | u_char lanesol; | | | |
| 1272 | u_char slotsol; | | | |
| 1273 | total_unit = dab_ptr->unit_count; | | | |
| 1274 | for (unitno = 0; unitno < total_unit; ++unitno) { | | | |
| 1275 | lanesol = dab_ptr->unit_location(unitno).lane_no; | | | |
| 1276 | slotsol = dab_ptr->unit_location(unitno).slot_no; | | | |
| 1277 | addr = LANE_0_START_ADDR + lanesol * LANE_ADDR_INC + | | | |
| 1278 | LANE_PEL_0_START_OFFSET + slotsol * LANE_PEL_ADDR_INC + | | | |
| 1279 | PEL_STATUS_CONTROL_OFFSET; | | | |
| 1280 | value = (u_short)read_loc_long((u_long *)addr); | | | |
| 1281 | DPRINTF(("aduai: la: %d sl: %d pel status: %04x\n", | | | |
| 1282 | lanesol, slotsol, value)); | | | |
| 1283 | if (((value & PEL_CS_ACTIVE_MASK) == 0) | | | |
| 1284 | ((value & PEL_CS_INITIALIZE_MASK) == 0) | | | |
| 1285 | ((value & PEL_CS_RESET_MASK) == 0)) | | | |
| 1286 | return(FALSE); | | | |
| 1287 | } | | | |
| 1288 | return(TRUE); | | | |
| 1289 | } | | | |
| 1290 | configure_pam(lanesol, lane_ptr) | | | |
| 1291 | u_char lanesol; | | | |
| 1292 | LANE_INFO *lane_ptr; | | | |
| 1293 | { | | | |
| 1294 | PAM_INFO *pam_ptr; | | | |
| 1295 | u_long board_id; | | | |
| 1296 | #ifdef MODELER | | | |
| 1297 | u_long addr; | | | |
| 1298 | u_long link_addr; | | | |
| 1299 | long i; | | | |
| 1300 | #endif | | | |
| 1301 | u_long total_mem; | | | |
| 1302 | } | | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/hwconfig.c

DATE

5/23/89

PAGE #

TIME

6:14:42 pm

12/70

LINE # SOURCE TEXT

```

1321 u_char    panno;
1322 u_char    temp;
1323 u_char    pam_count;
1324 u_char    config;
1325 u_char    pam_error_encountered;
1326 u_char    pam_type;
1327 u_char    cur_pam_type;
1328
1329 DPRINTF(("inside configure_pam\n"));
1330 cur_pam_type = PAC_PAM_2M;
1331
1332 (void)read_pam_count_reg(laseno, &pam_count);
1333 if (pam_count == 0) {
1334     lm_config_error(CERR_NO_PAM_FOR_PAC, laseno, 0);
1335     DPRINTF(("ERROR: no pam on lane %d\n", laseno));
1336     return(FAILURE);
1337 }
1338
1339 DPRINTF(("pam count: %d\n", pam_count));
1340
1341 /* find all of the PAM's in this lane */
1342 pam_error_encountered = 0;
1343 total_mem = 0;
1344 for (panno = 0; panno < pam_count; ++panno) {
1345     (void)read_pam_present(laseno, panno, &temp);
1346     if (temp) {
1347         DPRINTF(("found pam: %d\n", panno));
1348
1349         /* The PAM is present */
1350         pam_ptr = new_pam_info();
1351         lane_ptr->pam[panno] = pam_ptr;
1352         (void)read_pam_id(laseno, panno, &pam_type, &board_id);
1353
1354         pam_ptr->board_id = board_id;
1355
1356         /* Check to make sure that the PAM's are ordered according
1357          * to decreasing memory size.
1358          */
1359         if (pam_type > cur_pam_type) {
1360             lm_config_error(CERR_PAM_STACKED_WRONG, laseno, panno);
1361             DPRINTF(("ERROR: PAM %d is bigger than previous PAM\n",
1362                 panno));
1363             pam_error_encountered = 1;
1364         }
1365         else
1366             cur_pam_type = pam_type;
1367
1368         switch (pam_type) {
1369             case PAC_PAM_128K:
1370                 pam_ptr->mem_size = PAM128K_PTRN_COUNT;
1371                 break;
1372             case PAC_PAM_512K:
1373                 pam_ptr->mem_size = PAM512K_PTRN_COUNT;
1374                 break;
1375             case PAC_PAM_2M:
1376                 pam_ptr->mem_size = PAM2M_PTRN_COUNT;
1377                 break;
1378             default:
1379                 break;
1380         }
1381
1382         DPRINTF(("pam size: %d\n", pam_ptr->mem_size));
1383         total_mem += pam_ptr->mem_size;
1384     }
1385     else {
1386         lm_config_error(CERR_PAM_STRAPPED_WRONG, laseno, panno);
1387         DPRINTF(("ERROR: PAM %d on lane %d is strapped incorrectly\n",
1388             panno, laseno));
1389         pam_error_encountered = 1;
1390     }
1391 }
1392
1393 if (pam_error_encountered) {
1394     /* If there are any problems with PAM strapping, remove
1395     * all of the PAMs from the list.
1396     */
1397     for (panno = 0; panno < MAX_PAM_COUNT; ++panno)
1398         lane_ptr->pam[panno] = NULL;
1399     return(FAILURE);
1400 }
1401
1402 /* Write PAM config register */
1403 switch (pam_count) {
1404     case 1:
1405         switch (total_mem / PTRN_PER_BLOCK) {
1406             case BLOCKS_IN_128K:
1407                 config = 60;
1408                 break;
1409             case BLOCKS_IN_512K:
1410                 config = 61;
1411                 break;
1412             case BLOCKS_IN_2M:
1413                 config = 62;
1414                 break;
1415             default:
1416                 break;
1417         }
1418     break;
1419     case 2:
1420     case 3:
1421     case 4:
1422         config = ((total_mem / PTRN_PER_BLOCK) / BLOCKS_IN_128K) - 1;
1423         break;
1424     default:
1425         /* This condition will never occur */
1426         break;
1427 }
1428
1429 /* set ODD parity on LOW and HIGH word --> set the bit to 1 */
1430 config |= PAC_CONFIG_HIGH_WORD_PARITY_MASK |
1431          PAC_CONFIG_LOW_WORD_PARITY_MASK;
1432
1433 DPRINTF(("PAC config reg on lane: %d --> %08x\n", laseno, config));
1434
1435 /* Configure the PAC with EVEN parity, then clear the memory, and
1436 * then set it to ODD parity. This will catch software problems; i.e.
1437 * if the software reads uninitialized memory inadvertently.
1438 */

```

| Copyright 1989 Logic Modeling Systems | SOURCE PROGRAM lm1000/hwconfig.c | DATE 5/23/89 TIME 6:14:42 pm | PAGE # 13/71 |
|--|--|---------------------------------------|-----------------|
| LINE # | SOURCE TEXT | | |
| 1441 | write_loc_long((u_long *)(&lane_offset(laneno) + | | |
| 1442 | LANE_PAC_START_OFFSET + PAC_PAM_CONFIG_REG_OFFSET), | | |
| 1443 | (u_long)config); | | |
| 1444 | | | |
| 1445 | #ifdef MODELER | | |
| 1446 | addr = LANE_0_START_ADDR + laneno * LANE_ADDR_INC; | | |
| 1447 | link_addr = addr + LANE_LINK_TABLE_OFFSET; | | |
| 1448 | | | |
| 1449 | for (i = 0; i < total_num; ++i) { | | |
| 1450 | write_loc_long((u_long *)(&addr), (u_long)0); | | |
| 1451 | write_loc_long((u_long *)(&addr + LANE_SEGMENT_B_OFFSET), (u_long)0); | | |
| 1452 | write_loc_long((u_long *)(&addr + LANE_SEGMENT_C_OFFSET), (u_long)0); | | |
| 1453 | addr += PTHN_ADDR_INC; | | |
| 1454 | } | | |
| 1455 | | | |
| 1456 | for (i = 0; i < MAX_LINK_TABLE_COUNT; ++i) { | | |
| 1457 | write_loc_long((u_long *)(&link_addr), (u_long)0); | | |
| 1458 | link_addr += LINK_TABLE_ADDR_INC; | | |
| 1459 | } | | |
| 1460 | #else | | |
| 1461 | #ifdef DIRECTION | | |
| 1462 | dc_clear_ptr(laneno, (u_long)total_num); | | |
| 1463 | #endif | | |
| 1464 | #endif | | |
| 1465 | | | |
| 1466 | /* set ODD parity on LOW and HIGH word --> set the bit to 1 */ | | |
| 1467 | config = "PAC_CONFIG_HIGH_WORD_PARITY_MASK | | |
| 1468 | PAC_CONFIG_LOW_WORD_PARITY_MASK; | | |
| 1469 | | | |
| 1470 | write_loc_long((u_long *)(&lane_offset(laneno) + | | |
| 1471 | LANE_PAC_START_OFFSET + PAC_PAM_CONFIG_REG_OFFSET), | | |
| 1472 | (u_long)config); | | |
| 1473 | | | |
| 1474 | DPRINTF(("exiting configure_pam\n")); | | |
| 1475 | return(SUCCESS); | | |
| 1476 | } | | |
| 1477 | | | |
| 1478 | raise_pel_reset() | | |
| 1479 | { | | |
| 1480 | u_long magic_addr; | | |
| 1481 | u_long addr; | | |
| 1482 | u_short word; | | |
| 1483 | u_char laneno; | | |
| 1484 | u_char slotno; | | |
| 1485 | | | |
| 1486 | /* This routine doesnot RESET on all PELS which still have | | |
| 1487 | * reset asserted */ | | |
| 1488 | for (laneno = 0; laneno < MAX_LANE_COUNT; ++laneno) { | | |
| 1489 | | | |
| 1490 | for (slotno = 0; slotno < MAX_SLOT_COUNT; ++slotno) { | | |
| 1491 | if (system_config->lane[laneno]->pel[slotno] == NULL) | | |
| 1492 | continue; | | |
| 1493 | | | |
| 1494 | magic_addr = LANE_0_START_ADDR + laneno * LANE_ADDR_INC + | | |
| 1495 | LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC + | | |
| 1496 | PEL_NC_0_OFFSET; | | |
| 1497 | | | |
| 1498 | (void)read_loc_long((u_long *)(&magic_addr + NC_RESET_REG_OFFSET)); | | |
| 1499 | | | |
| 1500 | /* Need to read some other Magic Register to affect reset */ | | |
| 1501 | (void)read_loc_long((u_long *)(&magic_addr + NC_DATA_OUT_REG_OFFSET)); | | |
| 1502 | | | |
| 1503 | | | |
| 1504 | addr = LANE_0_START_ADDR + laneno * LANE_ADDR_INC + | | |
| 1505 | LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC + | | |
| 1506 | PEL_STATUS_CONTROL_OFFSET; | | |
| 1507 | word = read_loc_long((u_long *)(&addr)); | | |
| 1508 | DPRINTF(("rpr: la: %d sl: %d pel status: %04x\n", | | |
| 1509 | laneno, slotno, word)); | | |
| 1510 | | | |
| 1511 | if (word & "PEL_CS_RESET_MASK) | | |
| 1512 | continue; | | |
| 1513 | | | |
| 1514 | /* assert RESET */ | | |
| 1515 | word = PEL_CS_RESET_MASK & PEL_CS_IN_USE_LED_MASK; | | |
| 1516 | write_loc_long((u_long *)(&addr), (u_long)word); | | |
| 1517 | | | |
| 1518 | /* write INITIALIZE to 0 with RESET -- 1 */ | | |
| 1519 | word = (word & PEL_CS_INITIALIZE_MASK) "PEL_CS_RESET_MASK; | | |
| 1520 | write_loc_long((u_long *)(&addr), (u_long)word); | | |
| 1521 | | | |
| 1522 | | | |
| 1523 | } | | |
| 1524 | | | |
| 1525 | | | |
| 1526 | check_pel_stacking(dab_ptr, pel_location) | | |
| 1527 | DAB_INFO *dab_ptr; | | |
| 1528 | u_char pel_location[MAX_SLOT_COUNT]; | | |
| 1529 | { | | |
| 1530 | char i; | | |
| 1531 | u_char slots_occupied[MAX_LANE_COUNT][MAX_SLOT_COUNT]; | | |
| 1532 | char minlaneno; | | |
| 1533 | char maxlaneno; | | |
| 1534 | char minslotno; | | |
| 1535 | char maxslotno; | | |
| 1536 | char laneno; | | |
| 1537 | char slotno; | | |
| 1538 | | | |
| 1539 | minlaneno = MAX_LANE_COUNT; | | |
| 1540 | maxlaneno = -1; | | |
| 1541 | minslotno = MAX_SLOT_COUNT; | | |
| 1542 | maxslotno = -1; | | |
| 1543 | | | |
| 1544 | /* Initialize slots_occupied */ | | |
| 1545 | for (laneno = 0; laneno < MAX_LANE_COUNT; ++laneno) | | |
| 1546 | for (slotno = 0; slotno < MAX_SLOT_COUNT; ++slotno) | | |
| 1547 | slots_occupied[laneno][slotno] = 0; | | |
| 1548 | | | |
| 1549 | /* Mark all of the slots occupied by this device */ | | |
| 1550 | for (i = 0; i < dab_ptr->unit_count; ++i) { | | |
| 1551 | laneno = dab_ptr->unit_location[i].lane_no; | | |
| 1552 | slotno = dab_ptr->unit_location[i].slot_no; | | |
| 1553 | | | |
| 1554 | slots_occupied[laneno][slotno] = 1; | | |
| 1555 | | | |
| 1556 | if (laneno < minlaneno) | | |
| 1557 | minlaneno = laneno; | | |
| 1558 | if (laneno > maxlaneno) | | |
| 1559 | maxlaneno = laneno; | | |
| 1560 | if (slotno < minslotno) | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/hwconfig.c

DATE 5/23/89
TIME 6:14:42 pm

PAGE #
14/72

```

1561 minslotno = slotno;
1562 if (slotno > maxslotno)
1563     maxslotno = slotno;
1564 }
1565
1566 for (slotno = minslotno; slotno <= maxslotno; ++slotno) {
1567     /* Check if this slot is occupied at all */
1568     for (laneno = minlaneno; laneno <= maxlaneno; ++laneno) {
1569         if (slots_occupied[laneno][slotno] == 1)
1570             break;
1571     }
1572     /* Go to the next slot if there is no unit in this slot */
1573     if (laneno > maxlaneno)
1574         continue;
1575     /* Make sure that at this slotno there is a unit on each lane
1576      * that the device occupies. Don't do this check on
1577      * the rightmost slot.
1578      */
1579     if (slotno != maxslotno) {
1580         for (laneno = minlaneno; laneno <= maxlaneno; ++laneno) {
1581             if (dab_ptr->lane_used[laneno] == FALSE)
1582                 continue;
1583             if (slots_occupied[laneno][slotno] == 0)
1584                 return(FAILURE);
1585         }
1586     }
1587     /* Go through all other lanes occupied by this device and compare
1588      * the PEL stacking on this lane with the PEL stacking on the lane
1589      * found above.
1590      */
1591     for (laneno = minlaneno + 1; laneno <= maxlaneno; ++laneno) {
1592         if (dab_ptr->lane_used[laneno] == FALSE)
1593             continue;
1594         /* This condition can only happen for the rightmost slot */
1595         if (slots_occupied[laneno][slotno] == 0)
1596             continue;
1597         if (check_pel_to_left(pel_location, minlaneno,
1598                               laneno, slotno) == FAILURE) {
1599             return(FAILURE);
1600         }
1601     }
1602     return(SUCCESS);
1603 }
1604
1605 check_pel_to_left(pel_location, laneno1, laneno2, toslotno)
1606 u_char pel_location[[]MAX_SLOT_COUNT];
1607 char laneno1;
1608 char laneno2;
1609 char toslotno;
1610 {
1611     u_char slotno;
1612     DPRINTF(("inside check_pel_to_left on slot: %d between lane %d and %d\n",
1613             toslotno, laneno1, laneno2));
1614     for (slotno = 0; slotno < toslotno; ++slotno) {
1615         if (pel_location[laneno1][slotno] != pel_location[laneno2][slotno])
1616             return(FAILURE);
1617     }
1618     return(SUCCESS);
1619 }
1620
1621 init_magic_and_pel(laneno, slotno)
1622 u_char laneno;
1623 u_char slotno;
1624 {
1625     u_long magic_addr;
1626     DPRINTF(("inside init_magic_and_pel, lane: %d slot: %d\n",
1627             laneno, slotno));
1628     magic_addr = LANE_0_START_ADDR + laneno * LANE_ADDR_INC +
1629                 LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC +
1630                 PEL_MC_0_OFFSET;
1631     (void)read_loc_long((u_long *)(&magic_addr + MC_RESET_REG_OFFSET));
1632     /* Need to read some other Magic reg to affect reset */
1633     (void)read_loc_long((u_long *)(&magic_addr + MC_DATA_OUT_REG_OFFSET));
1634     (void)read_loc_long((u_long *)(&magic_addr));
1635     init_dab(laneno, slotno);
1636 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/initseq.c

DATE 5/23/89 PAGE #
TIME 6:14:43 pm 1/73

```

LINE # SOURCE TEXT
1  /* SCOS ID: initseq.c rev 3.1. 4/24/89 at 07:53:13 */
2  #include "device.h"
3  #include "message.h"
4  #include "hardware.h"
5  #include "osprun.h"
6  #include "lmserver.h"
7
8  #ifdef DEBUG
9  extern u_char loop_till_key;
10 extern u_long debug_key;
11 extern u_long debug_chan;
12 #endif
13
14 build_static_pattern_seq(instance)
15 INSTANCE_INFO *instance;
16
17 /* This routine builds the static pattern seq for an instance:
18 *   * PREAMBLE, PRE_FB_SEQ, FB_SEQ, POST_FB_SEQ
19 *   * by calling load_preamble(), load_pre_feedback_seq(),
20 *   * load_feedback_seq(), and load_post_feedback_seq(), respectively.
21 *   * Upon return from the routines load_preamble() and
22 *   * load_pre_feedback_seq() the INSTANCE.unit_addr is pointing to the
23 *   * last pattern loaded, whereas the INSTANCE.unit_addr is pointing at
24 *   * the next pattern address to be loaded upon return from
25 *   * load_feedback_seq() and load_post_feedback_seq().
26 *   * The following combinations of preamble.seq and reset.seq are allowed:
27 *   *   * PREAMBLE PRE FB POST
28 *   *   * PREAMBLE PRE FB
29 *   *   * PREAMBLE FB POST
30 *   *   * PREAMBLE FB
31 *   *   * PREAMBLE
32 *   *   * FB POST
33 *   *   * FB
34 *   * any other combinations are illegal and should be checked by the parser.
35 */
36
37 PREAMBLE preamble;
38 DAB_INFO *dab_ptr;
39 u_char lanes;
40 long dummy_count;
41 u_long *unit_addr;
42 u_long overflowed_patterns_count;
43 short i;
44 u_char reload_preamble_flag;
45
46 dab_ptr = dab_list(instance->dab_info_index);
47
48 setup_gbl_dummy_ptrs(dab_ptr);
49
50 if (allocate_initial_block(instance) == FAILURE)
51     return;
52
53 /* set the sequence start addresses on each lane */
54 for (lanes = 0; lanes < MAX_LANE_COUNT; ++lanes)
55     instance->seq_start_addr[lanes] =
56         instance->lane_addr[lanes].max_addr - BLOCK_ADDR_INC;
57
58 if (instance->definition->fb_seq_len != 0)
59     if (load_dummy_patterns(instance) == FAILURE)
60         return;
61
62 if (load_preamble(instance, (u_char)FALSE,
63     &preamble, &reload_preamble_flag) == FAILURE)
64     return;
65
66 if (instance->definition->pre_seq_len != 0) {
67     if (grow_patterns(instance) == FAILURE)
68         return;
69
70     if (load_pre_feedback_seq(instance) == FAILURE)
71         return;
72 }
73
74 if (instance->definition->fb_seq_len != 0) {
75     /* Add more dummy patterns to make sure that the feedback signal
76     * is quiet before we branch to the feedback sequence.
77     */
78     if (load_dummy_patterns2(instance) == FAILURE)
79         return;
80
81     if (allocate_feedback_block(instance) == FAILURE)
82         return;
83
84     if (load_feedback_seq(instance, &overflowed_patterns_count) == FAILURE)
85         return;
86
87     /* The sequence end bit has to be specified. SEQ_END_LATENCY patterns
88     * before the actual pattern we want to stop at. If the number of
89     * post feedback sequence is less than the SEQ_END_LATENCY, we need
90     * to add some dummy patterns in the beginning of the block after the
91     * feedback block.
92     */
93
94     /* Total number of lane patterns needed for various unit_count_per_lane
95     * (assuming SEQ_END_LATENCY = 6):
96     *   * unit_count_per_lane = 1 --> tot_lane_patterns_needed = 6
97     *   * unit_count_per_lane = 2 --> tot_lane_patterns_needed = 3
98     *   * unit_count_per_lane = 3 --> tot_lane_patterns_needed = 2
99     *   * unit_count_per_lane = 4 --> tot_lane_patterns_needed = 2
100    *   * unit_count_per_lane = 5 --> tot_lane_patterns_needed = 2
101    *   * unit_count_per_lane = 6 --> tot_lane_patterns_needed = 1
102    *   * unit_count_per_lane = 7 --> tot_lane_patterns_needed = 1
103    *   * .....
104    */
105
106    if (overflowed_patterns_count < SEQ_END_LATENCY) {
107        dummy_count = SEQ_END_LATENCY / dab_ptr->unit_count_per_lane;
108        if ((dummy_count * dab_ptr->unit_count_per_lane) < SEQ_END_LATENCY)
109            ++dummy_count;
110        dummy_count -= instance->definition->post_seq_len;
111    }
112    else
113        dummy_count = 0;
114
115    for (i = 0; i < dummy_count; ++i) {
116        write_pattern(instance,
117            instance->unit_addr(instance->cur_unit_addr_index)[0],
118            gbl_dummy_ptr);
119    }
120

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/initseq.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:43 pm | 2/74 |

```

LINE #          SOURCE TEXT
121      if (grow_pattern(instance) == FAILURE)
122      {
123      }
124      else {
125      /* load pre_feedback_seq() leaves the unit_addr pointing at the
126      * last pattern written. We need to increment it by one pattern
127      * so that the entry condition in load_post_feedback_seq() is
128      * the same whether we have feedback sequences or not.
129      */
130      if (grow_pattern(instance) == FAILURE)
131      {
132      return;
133      }
134      if (instance->definition->post_seq_len != 0)
135      if (load_post_feedback_seq(instance) == FAILURE)
136      {
137      return;
138      }
139      if (reload_preamble_flag == TRUE) {
140      if (reload_preamble(instance, &preamble) == FAILURE) {
141      free_preamble(&preamble);
142      return;
143      }
144      }
145      free_preamble(&preamble);
146      if (set_initial_values(instance) == FAILURE)
147      {
148      return;
149      }
150      /* Copy the current pattern_count to the static pattern count.
151      * Note that grow_pattern() was called last, so the number is the
152      * pattern_count is 1 logical pattern more than what it should be.
153      */
154      instance->static_pattern_count = instance->pattern_count -
155      dab_ptr->unit_count_per_lane;
156      /* Copy the unit_addr to first_user_ptrn_unit_addr */
157      temp_unit_addr = instance->unit_addr[instance->cur_unit_addr_index][0];
158      for (i = 0; i < dab_ptr->unit_count; ++i)
159      instance->first_user_ptrn_unit_addr[i] = temp_unit_addr[i];
160      }
161      load_preamble(instance, preamble_for_power_up, preamble, reload_preamble_flag)
162      INSTANCE_INFO *instance;
163      u_char preamble_for_power_up;
164      PREAMBLE *preamble;
165      u_char *reload_preamble_flag;
166      {
167      PREAMBLE_WORD *preamble_word;
168      PTRN_BITS LONGWORD *unit_ptrn;
169      PTRN_BITS LONGWORD *preamble_unit_ptrn;
170      PTRN_BITS LONGWORD *ptrn_bits_ptr;
171      PIN_SPEC *pin_spec_ptr;
172      DAB_INFO *dab_ptr;
173      DEVICE_SPEC *dev_ptr;
174      EXTRA_DEVICE_SPEC *extra_dev_ptr;
175      SEQ_SPEC *seq_spec_ptr;
176      PTRN_BITS LONGWORD *ident_outputs_ptr;
177      PTRN_BITS LONGWORD *ident_ios_ptr;
178      u_long *unit_ptr;
179      u_long *temp_unit_addr;
180      u_short seq_count;
181      u_short pin_count;
182      u_short pinno;
183      u_short unitno;
184      u_char wordno;
185      u_char hitno;
186      u_char unit_count;
187      u_char i;
188      u_char j;
189      u_char temp;
190      DPRINTF(("inside load_preamble\n"));
191      *reload_preamble_flag = FALSE;
192      def_ptr = instance->definition;
193      extra_dev_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
194      dab_ptr = dab_list[instance->dab_info_index];
195      unit_count = dab_ptr->unit_count;
196      preamble_word = (PREAMBLE_WORD *)preamble;
197      for (i = 0; i < sizeof(PREAMBLE) / sizeof(PTRN_BITS *); ++i) {
198      preamble_word->word[i] =
199      (PTRN_BITS *)DCALLOC((unsigned)unit_count,
200      (unsigned)sizeof(PTRN_BITS));
201      if (preamble_word->word[i] == NULL) {
202      is_queue_message(ERROR_MSG, "out of memory on modeler");
203      for (j = 0; j < i; ++j) {
204      DFREE((char *)preamble_word->word[j]);
205      }
206      return(FAILURE);
207      }
208      }
209      for (i = 0; i < unit_count; ++i) {
210      unit_ptrn = (PTRN_BITS LONGWORD *)preamble->adlen3b[i];
211      unit_ptrn->word[0] = 0xffffffff;
212      unit_ptrn->word[1] = 0xffffffff;
213      unit_ptrn->word[2] = 0xffffffff;
214      unit_ptrn = (PTRN_BITS LONGWORD *)preamble->adlen2b[i];
215      unit_ptrn->word[0] = 0xffffffff;
216      unit_ptrn->word[1] = 0xffffffff;
217      unit_ptrn->word[2] = 0xffffffff;
218      unit_ptrn = (PTRN_BITS LONGWORD *)preamble->adlen1b[i];
219      unit_ptrn->word[0] = 0xffffffff;
220      unit_ptrn->word[1] = 0xffffffff;
221      unit_ptrn->word[2] = 0xffffffff;
222      unit_ptrn = (PTRN_BITS LONGWORD *)preamble->seqqdb[i];
223      unit_ptrn->word[0] = 0xffffffff;
224      unit_ptrn->word[1] = 0xffffffff;
225      unit_ptrn->word[2] = 0xffffffff;
226      unit_ptrn = (PTRN_BITS LONGWORD *)preamble->lcychdb[i];
227      unit_ptrn->word[0] = 0xffffffff;

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/initseq.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:43 pm | 3/75 |

```

LINE #
241  unit_ptr->word[1] = 0xffffffff;
242  unit_ptr->word[2] = 0xffff0000;
243  unit_ptr = (PTRN_BITS_LONGWORD *) &preamble->lcymdb[1];
244  unit_ptr->word[0] = 0xffffffff;
245  unit_ptr->word[1] = 0xffffffff;
246  unit_ptr->word[2] = 0xffff0000;
247  unit_ptr = (PTRN_BITS_LONGWORD *) &preamble->format1[1];
248  unit_ptr->word[0] = 0xffffffff;
249  unit_ptr->word[1] = 0xffffffff;
250  unit_ptr->word[2] = 0xffff0000;
251  unit_ptr = (PTRN_BITS_LONGWORD *) &preamble->format0[1];
252  unit_ptr->word[0] = 0xffffffff;
253  unit_ptr->word[1] = 0xffffffff;
254  unit_ptr->word[2] = 0xffff0000;
255  unit_ptr = (PTRN_BITS_LONGWORD *) &preamble->dummy_hmdeb[1];
256  unit_ptr->word[0] = 0xffffffff;
257  unit_ptr->word[1] = 0xffffffff;
258  unit_ptr->word[2] = 0xffff0000;
259  unit_ptr = (PTRN_BITS_LONGWORD *) &preamble->hmdeb[1];
260  unit_ptr->word[0] = 0xffffffff;
261  unit_ptr->word[1] = 0xffffffff;
262  unit_ptr->word[2] = 0xffff0000;
263  unit_ptr = (PTRN_BITS_LONGWORD *) &preamble->dummy_data2[1];
264  unit_ptr->word[0] = 0xffffffff;
265  unit_ptr->word[1] = 0xffffffff;
266  unit_ptr->word[2] = 0xffff0000;
267  }
268
269  pin_count = def_ptr->pin_cnt;
270  pin_spec_ptr = &def_ptr->pin_table[0];
271  for (pinno = 0; pinno < pin_count; ++pinno) {
272
273      if (pin_spec_ptr->direction == NONE) {
274          ++pin_spec_ptr;
275          continue;
276      }
277
278      if ((pin_spec_ptr->direction == POWER) ||
279          (pin_spec_ptr->direction == GROUND) ||
280          (pin_spec_ptr->direction == NC)) {
281
282          /* Float these pins */
283
284          word_ptr = &pin_spec_ptr->word;
285          unitno = word_ptr->unitno;
286          wordno = word_ptr->wordno;
287          bitno = word_ptr->bitno;
288
289          set_ptrn_bit(&preamble->seqhdb [unitno], wordno, bitno);
290          set_ptrn_bit(&preamble->adlshdb [unitno], wordno, bitno);
291
292          ++pin_spec_ptr;
293          continue;
294      }
295
296      /* INPUT, OUTPUT, or IO pin */
297
298      word_ptr = &pin_spec_ptr->word;
299      unitno = word_ptr->unitno;
300      wordno = word_ptr->wordno;
301      bitno = word_ptr->bitno;
302
303      /* HUBBIS */
304      if (pin_spec_ptr->h_drive == ALWAYS_ON)
305          set_ptrn_bit(&preamble->hddiab[unitno], wordno, bitno);
306
307      /* For I/O STORE pin in NON ULTRA FAST mode, set HUBBIS to 1.
308       * We can tell if it's in Non-Ultra Fast mode because INSEQED
309       * will be off.
310       */
311      if ((pin_spec_ptr->direction == IO) &&
312          (pin_spec_ptr->pin_class == STORE) &&
313          (pin_spec_ptr->pin_seq_drive != H_DRIVE) &&
314          (pin_spec_ptr->pin_seq_drive != HM_DRIVE)) {
315          set_ptrn_bit(&preamble->hddiab[unitno], wordno, bitno);
316      }
317
318      /* Set DOFF to 1 (edge 5) for I/O STORE pins */
319      if ((pin_spec_ptr->direction == IO) &&
320          (pin_spec_ptr->pin_class == STORE))
321          set_ptrn_bit(&preamble->doff[unitno], wordno, bitno);
322
323      /* HUBBIS */
324      if (pin_spec_ptr->h_drive == DRIVE_ON)
325          reset_ptrn_bit(&preamble->hmdenb[unitno], wordno, bitno);
326
327      /* HUBBIS and SEQUEN */
328      switch (pin_spec_ptr->pin_seq_drive) {
329      case NO_DRIVE:
330          set_ptrn_bit(&preamble->seqhdb [unitno], wordno, bitno);
331          break;
332      case H_DRIVE:
333          reset_ptrn_bit(&preamble->seqhdb [unitno], wordno, bitno);
334          set_ptrn_bit(&preamble->seqhdb [unitno], wordno, bitno);
335          break;
336      case M_DRIVE:
337          /* Default is Medium drive */
338          break;
339      case HM_DRIVE:
340          reset_ptrn_bit(&preamble->seqhdb [unitno], wordno, bitno);
341          break;
342      default:
343          break;
344      }
345
346      /* LCYCHDB and LCTCHDB */
347      switch (pin_spec_ptr->last_cyc_drive) {
348      case NO_DRIVE:
349          break;
350      case H_DRIVE:
351          reset_ptrn_bit(&preamble->lcychdb [unitno], wordno, bitno);
352          break;
353      case M_DRIVE:
354          reset_ptrn_bit(&preamble->lcychdb [unitno], wordno, bitno);
355          break;
356      case HM_DRIVE:
357          reset_ptrn_bit(&preamble->lcychdb [unitno], wordno, bitno);
358          reset_ptrn_bit(&preamble->lcychdb [unitno], wordno, bitno);
359          break;
360      default:

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/initseq.c

DATE 5/23/89
TIME 6:14:43 pm

PAGE #
4/76

LINE # SOURCE TEXT

```

361     break;
362 }
363
364 /* FORWARD[1-0], SET[1-0], RESET[1-0] */
365 switch (pin_spec_ptr->clk_format) {
366     case DNRZ:
367         if (pin_spec_ptr->pin_class == STORE) {
368             /* use edge #1:
369              * SET [1-0] <- 1
370              * RESET [1-0] <- 3
371              */
372             set_ptrn_bit(&preamble->set0 [unitno], wordno, bitno);
373             set_ptrn_bit(&preamble->reset1 [unitno], wordno, bitno);
374             set_ptrn_bit(&preamble->reset0 [unitno], wordno, bitno);
375         }
376         else {
377             /* use edge #2:
378              * SET [1-0] <- 0
379              * RESET [1-0] <- 2
380              */
381             set_ptrn_bit(&preamble->reset1 [unitno], wordno, bitno);
382         }
383     }
384     break;
385
386     case R0:
387         reset_ptrn_bit(&preamble->format1 [unitno], wordno, bitno);
388
389         /* falling edge (significant edge) --> use edge #3
390          * rising edge (non significant edge) --> use edge #4
391          */
392         if (temp & 0x1) {
393             SET [1-0] <- 2
394             RESET [1-0] <- 3
395         }
396         else {
397             SET [1-0] <- 3
398             RESET [1-0] <- 2
399         }
400         set_ptrn_bit(&preamble->set1 [unitno], wordno, bitno);
401         set_ptrn_bit(&preamble->reset1 [unitno], wordno, bitno);
402         set_ptrn_bit(&preamble->reset0 [unitno], wordno, bitno);
403     }
404     break;
405
406     case R1:
407         /* make sure the clock does not toggle in dummy_data */
408         set_ptrn_bit(&preamble->dummy_data [unitno], wordno, bitno);
409         set_ptrn_bit(&preamble->dummy_data2 [unitno], wordno, bitno);
410         reset_ptrn_bit(&preamble->format0 [unitno], wordno, bitno);
411
412         /* rising edge (significant edge) --> use edge #3
413          * falling edge (non significant edge) --> use edge #1
414          */
415         if (temp & 0x1) {
416             SET [1-0] <- 1
417             RESET [1-0] <- 1
418         }
419         else {
420             SET [1-0] <- 1
421             RESET [1-0] <- 1
422         }
423         set_ptrn_bit(&preamble->set0 [unitno], wordno, bitno);
424         set_ptrn_bit(&preamble->reset0 [unitno], wordno, bitno);
425     }
426     break;
427
428     case RC:
429         /* ??? not supported */
430         set_ptrn_bit(&preamble->reset1 [unitno], wordno, bitno);
431     }
432     break;
433
434     default:
435         break;
436 }
437
438 /* SDLEN[3-0] */
439 temp = pin_spec_ptr->s_drive_hi;
440 DPRINTF(("s_drive_high: %d\n", temp));
441 if (temp & 0x8) {
442     set_ptrn_bit(&preamble->adlen3 [unitno], wordno, bitno);
443 }
444 else {
445     reset_ptrn_bit(&preamble->adlen3 [unitno], wordno, bitno);
446 }
447 if (temp & 0x4) {
448     set_ptrn_bit(&preamble->adlen2 [unitno], wordno, bitno);
449 }
450 else {
451     reset_ptrn_bit(&preamble->adlen2 [unitno], wordno, bitno);
452 }
453 if (temp & 0x2) {
454     set_ptrn_bit(&preamble->adlen1 [unitno], wordno, bitno);
455 }
456 else {
457     reset_ptrn_bit(&preamble->adlen1 [unitno], wordno, bitno);
458 }
459 if (temp & 0x1) {
460     set_ptrn_bit(&preamble->adlen0 [unitno], wordno, bitno);
461 }
462 else {
463     reset_ptrn_bit(&preamble->adlen0 [unitno], wordno, bitno);
464 }
465
466 /* SDLEN[3-0]B */
467 temp = pin_spec_ptr->s_drive_low;
468 DPRINTF(("s_drive_low: %d\n", temp));
469 if (temp & 0x8) {
470     set_ptrn_bit(&preamble->adlen3b [unitno], wordno, bitno);
471 }
472 else {
473     reset_ptrn_bit(&preamble->adlen3b [unitno], wordno, bitno);
474 }
475 if (temp & 0x4) {
476     set_ptrn_bit(&preamble->adlen2b [unitno], wordno, bitno);
477 }
478 else {
479     reset_ptrn_bit(&preamble->adlen2b [unitno], wordno, bitno);
480 }
481 if (temp & 0x2) {
482     set_ptrn_bit(&preamble->adlen1b [unitno], wordno, bitno);
483 }
484 else {
485     reset_ptrn_bit(&preamble->adlen1b [unitno], wordno, bitno);
486 }
487 if (temp & 0x1) {
488     set_ptrn_bit(&preamble->adlen0b [unitno], wordno, bitno);
489 }
490 else {
491     reset_ptrn_bit(&preamble->adlen0b [unitno], wordno, bitno);
492 }
493
494 /* Simulate pull up on feedback pin */
495 if (preamble_for_power_up == FALSE) {
496     if (pin_spec_ptr->feedback == 1) {
497         if (pin_spec_ptr->direction == IO) {
498             if (pin_spec_ptr->in_seq_drive == M_DRIVE) {
499                 set_ptrn_bit(&preamble->seqfdb [unitno], wordno, bitno);
500                 reset_ptrn_bit(&preamble->seqfdb [unitno], wordno, bitno);
501                 *reload_preamble_flag = TRUE;
502             }
503         }
504         else if (pin_spec_ptr->direction == OUT) {
505             reset_ptrn_bit(&preamble->seqfdb [unitno], wordno, bitno);
506             *reload_preamble_flag = TRUE;
507         }
508     }
509 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/initseq.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:43 pm | 5/77 |

```

481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600

```

}
 }
 ++pin_spec_ptr;
 }
 /* Setup the pattern control bits */
 for (i = 0; i < sizeof(PREAMBLE) / sizeof(PTRN_BITS *), ++i) {
 ptrn_bits_ptr = preamble.word->word[i];
 for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
 set_pel_ctl(&ptrn_bits_ptr->ctl, PEL_CTL_LOAD_CONTROL);
 set_pel_sel(&ptrn_bits_ptr->ctl, dab_ptr->unit_location[unitno].slot_no);
 ++ptrn_bits_ptr;
 }
 }
 /* modify the PEL CONTROL of SELECT_PEL */
 ptrn_bits_ptr = preamble->select_pel;
 for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
 set_pel_ctl(&ptrn_bits_ptr->ctl, PEL_CTL_SELECT_PEL);
 ++ptrn_bits_ptr;
 }
 /* modify the PEL CONTROL of LOAD_EDEN */
 ptrn_bits_ptr = preamble->load_edens;
 for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
 set_pel_ctl(&ptrn_bits_ptr->ctl, PEL_CTL_LOAD_EDEN_FORMAT);
 ++ptrn_bits_ptr;
 }
 /* modify the PEL CONTROL of DUMMY_EDENB */
 ptrn_bits_ptr = preamble->dummy_hndensb;
 for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
 set_pel_ctl(&ptrn_bits_ptr->ctl, PEL_CTL_LOAD_EDEN);
 ++ptrn_bits_ptr;
 }
 /* modify the PEL CONTROL of EDENB */
 ptrn_bits_ptr = preamble->hndensb;
 for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
 set_pel_ctl(&ptrn_bits_ptr->ctl, PEL_CTL_LOAD_EDEN);
 ++ptrn_bits_ptr;
 }
 /* modify the PEL CONTROL of DUMMY_DATA */
 ptrn_bits_ptr = preamble->dummy_data;
 for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
 if (dab_ptr->unit_location[unitno].last_is_lane)
 set_pel_ctl(&ptrn_bits_ptr->ctl, PEL_CTL_LOAD_DATA_LAST_UNIT);
 else
 set_pel_ctl(&ptrn_bits_ptr->ctl, PEL_CTL_LOAD_DATA);
 ++ptrn_bits_ptr;
 }
 /* modify the PEL CONTROL of DUMMY_DATA2 */
 ptrn_bits_ptr = preamble->dummy_data2;
 for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
 if (dab_ptr->unit_location[unitno].last_is_lane)
 set_pel_ctl(&ptrn_bits_ptr->ctl, PEL_CTL_LOAD_DATA_LAST_UNIT);
 else
 set_pel_ctl(&ptrn_bits_ptr->ctl, PEL_CTL_LOAD_DATA);
 ++ptrn_bits_ptr;
 }
 }
 /* Save the PREAMBLE LITCHES, LITCHES, and EDENB for the instance */
 for (i = 0; i < sizeof(PREAMBLE) / sizeof(PTRN_BITS *), ++i) {
 if (&preamble_word->word[i] == &preamble->hndensb) {
 /* copy the EDENB ptr into hndensb loaded in instance info */
 unit_ptrn = (PTRN_BITS_LONGWORD *)instance->hndensb_loaded;
 preamble_unit_ptrn =
 (PTRN_BITS_LONGWORD *)preamble_word->word[i];
 ident_outputs_ptr = extra_def_ptr->ident_outputs;
 ident_ios_ptr = extra_def_ptr->ident_ios;
 for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
 for (wordno = 0; wordno < 3; ++wordno) {
 unit_ptrn[unitno].word[wordno] =
 preamble_unit_ptrn[unitno].word[wordno];
 }
 /* Turn off the EDENB for OUTPUT and IO pins */
 preamble_unit_ptrn[unitno].word[wordno] |=
 (ident_outputs_ptr[unitno].word[wordno] |
 ident_ios_ptr[unitno].word[wordno]);
 }
 }
 }
 /* Turn back on the EDENB for those pins specified in either the
 * power-up reset sequence or the normal reset sequence.
 */
 if (preamble_for_power_up == TRUE) {
 in_queue_message(ERROR_MSG, "internal error: can't have power up sequence");
 return(FAILURE);
 }
 else {
 seq_count = def_ptr->seq_cnt;
 for (pin_count = 0; pin_count < seq_count; ++pin_count) {
 seq_spec_ptr = def_ptr->seq_table[pin_count];
 pinno = seq_spec_ptr->pin_number;
 wvb_ptr = spe_to_short_offset(pinno);
 unitno = wvb_ptr->unitno;
 wordno = wvb_ptr->wordno;
 bitno = wvb_ptr->bitno;
 reset_ptrn_bit(&preamble_unit_ptrn[unitno], wordno, bitno);
 }
 }
 }
 }
 }

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/initseq.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:43 pm | 6/78 |

```

LINE # SOURCE TEXT
601
602 if (!preamble_word->word[i] == &preamble->lcychdb) {
603
604     unit_ptrn = (PTRN_BITS_LONGWORD *)instance->lcychdb_loaded;
605     preamble_unit_ptrn =
606         (PTRN_BITS_LONGWORD *)preamble_word->word[i];
607
608     temp_unit_addr = instance->
609         unit_addr(instance->cur_unit_addr_index)[0];
610
611     for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
612
613         /* copy the addr of LITCHES into lcychdb_addr in instance_info */
614         instance->lcychdb_addr[unitno] = temp_unit_addr(unitno);
615
616         /* copy the LITCHES pattern into lcychdb_loaded in instance_info */
617         for (wordno = 0; wordno < 3; ++wordno) {
618             unit_ptrn[unitno].word[wordno] =
619                 preamble_unit_ptrn[unitno].word[wordno];
620         }
621     }
622
623
624
625 if (!preamble_word->word[i] == &preamble->lcycmdb) {
626
627     unit_ptrn = (PTRN_BITS_LONGWORD *)instance->lcycmdb_loaded;
628     preamble_unit_ptrn =
629         (PTRN_BITS_LONGWORD *)preamble_word->word[i];
630
631     temp_unit_addr = instance->
632         unit_addr(instance->cur_unit_addr_index)[0];
633
634     for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
635
636         /* copy the addr of LITCHES into lcycmdb_addr in instance_info */
637         instance->lcycmdb_addr[unitno] = temp_unit_addr(unitno);
638
639         /* copy the LITCHES pattern into lcycmdb_loaded in instance_info */
640         for (wordno = 0; wordno < 3; ++wordno) {
641             unit_ptrn[unitno].word[wordno] =
642                 preamble_unit_ptrn[unitno].word[wordno];
643         }
644     }
645
646
647
648     write_patterns(instance,
649         instance->unit_addr(instance->cur_unit_addr_index)[0],
650         preamble_word->word[i]);
651
652     /* Don't grow the pattern on the last preamble pattern,
653      * because otherwise grow_patterns() might allocate new blocks which
654      * will have to be retracted if the preamble is followed by feedback
655      * sequence (NO pre feedback sequence).
656      */
657     if (1 != ((sizeof(PREAMBLE) / sizeof(PTRN_BITS *)) - 1))
658         if (grow_patterns(instance) == FAILURE)
659             return(FAILURE);
660
661     return(SUCCESS);
662 }
663
664
665 free_preamble(preamble);
666 PREAMBLE = preamble;
667
668 PREAMBLE_WORD = preamble_word;
669 u_char
670     1;
671
672 preamble_word = (PREAMBLE_WORD *)preamble;
673
674 for (i = 0; i < sizeof(PREAMBLE) / sizeof(PTRN_BITS *); ++i)
675     FREE((char *)preamble_word->word[i]);
676
677 reload_preamble(instance, preamble)
678 INSTANCE_INFO *instance,
679 PREAMBLE *preamble,
680 {
681     PREAMBLE_WORD *preamble_word;
682     PTRN_BITS_LONGWORD *unit_ptrn;
683     PTRN_BITS_LONGWORD *preamble_unit_ptrn;
684     PIN_SPEC *pin_spec_ptr;
685     UWB_OFFSET *uwb_ptr;
686     DEVICE_SPEC *dev_ptr;
687     DAB_INFO *dab_ptr;
688     u_long pin_count;
689     u_short unitno;
690     u_short pinno;
691     u_char wordno;
692     u_char bitno;
693     i;
694
695
696
697     DPRINTF(("inside reload_preamble\n"));
698     dev_ptr = instance->definition;
699     dab_ptr = dab_list(instance->dab_info_index);
700
701     preamble_word = (PREAMBLE_WORD *)preamble;
702
703     /* Load a dummy pattern after the feedback/post-feedback sequence
704      * to let the last DATA pattern to get out of the MAGIC chip
705      * before reloading the preamble.
706      */
707     setup_gbl_dummy_ptrn(dab_ptr);
708     write_patterns(instance,
709         instance->unit_addr(instance->cur_unit_addr_index)[0],
710         gbl_dummy_ptrn);
711     if (grow_patterns(instance) == FAILURE)
712         return(FAILURE);
713
714     pin_count = dev_ptr->pin_cnt;
715     pin_spec_ptr = dev_ptr->pin_table[0];
716     for (pinno = 0; pinno < pin_count; ++pinno) {
717         if ((pin_spec_ptr->direction == NONE) ||
718             (pin_spec_ptr->direction == POWER) ||
719             (pin_spec_ptr->direction == GROUND))
720

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/initseq.c | DATE 5/23/89 TIME 6:14:43 pm | PAGE # 7/79 |
|--|--|------------------------------------|---------------------------------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 721 | <pre> (pin_spec_ptr->direction == MC) (pin_spec_ptr->direction == IN)) { *pin_spec_ptr; continue; } wdb_ptr = tpe_to_short_offset(pinspec); unitno = wdb_ptr->unitno; wordno = wdb_ptr->wordno; bitno = wdb_ptr->bitno; 722 723 724 725 /* SWITCHES and SEQUENCES */ switch (pin_spec_ptr->is_seq_drive) { 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 </pre> | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/initseq.c

DATE 5/23/89
TIME 6:14:43 pm

PAGE #
8/80

```

LINE # SOURCE TEXT
841
842 DPRINTF(("inside load_pre_feedback_seq\n"));
843
844 def = instance->definition;
845
846 pre_seq_length = def->pre_seq_len;
847 seq_count = def->seq_cnt;
848
849 for (i = 0; i < pre_seq_length; ++i) {
850
851     word_offset = i / 8;
852     bit_offset = 7 - i % 8;
853
854     for (pin_count = 0; pin_count < seq_count; ++pin_count) {
855
856         seq_spec_ptr = def->seq_table[pin_count];
857         bit_array = seq_spec_ptr->pre_bits;
858         pin_number = seq_spec_ptr->pin_number;
859
860         word_ptr = seq_ptr + word_offset * pin_number;
861         wordno = word_ptr->wordno;
862         bitno = word_ptr->bitno;
863
864         if ((bit_array[word_offset] >> bit_offset) & 1) {
865             set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
866         }
867         else {
868             reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
869         }
870     }
871
872     write_patterns(instance,
873                   instance->unit_addr(instance->cur_unit_addr_index)[0],
874                   instance->ptrn_loaded);
875
876     /* Don't grow the pattern on the last pre feedback sequence pattern.
877      * because otherwise grow_patterns() might allocate new blocks which
878      * will have to be retracted if the pre feedback is followed by
879      * a feedback sequence.
880     */
881     if (i != (pre_seq_length - 1))
882         if (grow_patterns(instance) == FAILURE)
883             return(FAILURE);
884 }
885 return(SUCCESS);
886 }
887
888 load_feedback_seq(instance, overflowed_patterns_count)
889 INSTANCE_INFO *instance;
890 u_long overflowed_patterns_count;
891 {
892     DEVICE_SPEC def;
893     PTRN_BITS_LONGWORD dummy_long;
894     SEQ_SPEC seq_spec_ptr;
895     DAB_INFO dab_ptr;
896     WORD_OFFSET word_ptr;
897     char *bit_array;
898     u_long dest_addr;
899     u_long source_addr[MAX_LANE_COUNT];
900     u_long saved_block_addr[MAX_LANE_COUNT];
901     u_long temp_unit_addr;
902     short leftover;
903     u_short quiet_patterns;
904     short added_patterns;
905     short extra_patterns;
906     u_char branch_sop;
907     u_short dummy_count;
908     short branch_delay;
909     short branch_location;
910     short physical_seq_len;
911     short sub_sequence;
912     u_short pin_number;
913     u_short fb_ptrn_count;
914     short number_of_copies;
915     short bits_per_pin = 1;
916     u_short i;
917     u_short j;
918     u_short played;
919     u_short ptrn_to_copy;
920     short addr_offset;
921     u_short word_offset;
922     u_short pin_count;
923     u_short seq_count;
924     u_short fb_seq_length;
925     u_short power_of_2_ceiling;
926     short dummy_count1;
927     short dummy_count2;
928     u_char total_unit;
929     u_char unitno;
930     u_char wordno;
931     u_char bitno;
932     u_char bit_offset;
933     u_char lane;
934     u_char max_block_size;
935
936 DPRINTF(("inside load_feedback\n"));
937
938 def = instance->definition;
939
940 dab_ptr = dab_list(instance->dab_info_index);
941
942 fb_ptrn_count = 0;
943 if (bits_per_pin == 1) {
944     if ((def->fb_seq_len + dab_ptr->unit_count_per_lane) >
945         (FB_PTRN_COUNT - MAX_DUMMY_COUNT_1_BIT_PER_PIN)) {
946         ln_queue_message(ERROR_MSG, "feedback sequence too long, max length: %d",
947                         (FB_PTRN_COUNT - MAX_DUMMY_COUNT_1_BIT_PER_PIN) /
948                         dab_ptr->unit_count_per_lane);
949         return(FAILURE);
950     }
951 }
952 else {
953     if ((def->fb_seq_len + dab_ptr->unit_count_per_lane * 2) >
954         (FB_PTRN_COUNT - MAX_DUMMY_COUNT_2_BIT_PER_PIN)) {
955         ln_queue_message(ERROR_MSG, "feedback sequence too long, max length: %d",
956                         (FB_PTRN_COUNT - MAX_DUMMY_COUNT_2_BIT_PER_PIN) /
957                         (dab_ptr->unit_count_per_lane * 2));
958         return(FAILURE);
959     }
960 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/initseq.c

DATE 5/23/89
TIME 6:14:43 pm

PAGE #
9/81

```

LINE # SOURCE TEXT
961
962 fb_seq_length = dcb->fb_seq_len;
963 seq_count = dcb->seq_cnt;
964
965 for (i = 0; i < fb_seq_length; ++i) {
966     word_offset = i / 8;
967     bit_offset = 7 - i % 8;
968
969     for (pin_count = 0; pin_count < seq_count; ++pin_count) {
970         seq_spec_ptr = dcb->seq_table[pin_count];
971         bit_array = seq_spec_ptr->fb_bits;
972         pin_number = seq_spec_ptr->pin_number;
973
974         uwb_ptr = dcb->uwb_offset[pin_number];
975         unitno = uwb_ptr->unitno;
976         wordno = uwb_ptr->wordno;
977         bitno = uwb_ptr->bitno;
978
979         if ((bit_array[word_offset] >> bit_offset) & 1) {
980             set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
981         }
982         else {
983             reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
984         }
985     }
986
987     fb_ptrn_count += dcb_ptr->unit_count_per_lane;
988     write_ptrn(instance,
989         instance->unit_addr(instance->cur_unit_addr_index)[0],
990         instance->ptrn_loaded);
991     if (grow_ptrn(instance) == FAILURE)
992         return(FAILURE);
993 }
994
995 physical_seq_len = fb_ptrn_count;
996
997 /* Up to this point the pattern address has been incremented by
998 * unit_count_per_lane units worth of addresses. From now on we
999 * are going to start writing dummy patterns which is only 1 unit
1000 * long with write_ptrn_unit(). The write_ptrn_unit() will
1001 * look at the LANE_ADDR_INFO.last_unit_addr for the address to
1002 * write to. Since the last call to grow_ptrn has incremented
1003 * the address for unit_count_per_lane units, we need to adjust
1004 * it by calling the adjust_ptrn_addr().
1005 */
1006 adjust_ptrn_addr(instance);
1007
1008 /* We need to wait the following number of PCLK:
1009 * 4 PCLK for the propagation delay from DUT output
1010 * to the input of the synchronizer register.
1011 * Later: this should depend on the frequency
1012 * we are using.
1013 * 2 logical clock since we assume that the DUT output changes
1014 * within 2 logical clock after the input.
1015 * Later: make this delay specifiable by the
1016 * user in the DADDEF.
1017 * 1 logical clock since the clock edges appear for the whole
1018 * logical clock after the pattern is latched
1019 * into the MAGIC register.
1020 */
1021 quiet_patterns = 4 + 3 * (dcb_ptr->unit_count_per_lane * bits_per_pin);
1022
1023 /* The clock edges can only span 3 PCLK maximum. So if
1024 * (unit_count_per_lane * bits_per_pin) is greater than 3 then we can
1025 * actually reduce the quiet_patterns_count by the remainder of PCLK where
1026 * there are no edges.
1027 */
1028 extra_patterns = dcb_ptr->unit_count_per_lane * bits_per_pin - 3;
1029 if (extra_patterns < 0)
1030     extra_patterns = 0;
1031
1032 added_patterns = quiet_patterns - extra_patterns;
1033 if (added_patterns < 0)
1034     added_patterns = 0;
1035
1036 /* Add 1 more pattern to make the subsequence pattern even, so that
1037 * the branch command will stay at the odd address.
1038 */
1039 if (even(physical_seq_len + added_patterns))
1040     branch_nop = 0;
1041 else
1042     branch_nop = 1;
1043
1044 dummy_count = added_patterns + branch_nop;
1045 DPRINTF(("FB dummy_count: %d\n", dummy_count));
1046
1047 for (i = 0; i < dummy_count; ++i) {
1048     ++fb_ptrn_count;
1049     write_ptrn_unit(instance, dcb_ptr->dummy_ptrn);
1050     if (grow_ptrn(instance) == FAILURE)
1051         return(FAILURE);
1052 }
1053
1054 sub_sequence = physical_seq_len + added_patterns + branch_nop;
1055 branch_delay = physical_seq_len - extra_patterns +
1056     quiet_patterns + FB_PIPE_DELAY;
1057
1058 if (!even(branch_delay))
1059     ++branch_delay;
1060
1061 branch_location = branch_delay % sub_sequence - 1;
1062
1063 if (branch_location < 0) {
1064     if (branch_location == -1) {
1065         branch_location = sub_sequence - 1;
1066     }
1067     else {
1068         log_queue_message(ERROR_MSG, "internal error: branch location is negative");
1069         return(FAILURE);
1070     }
1071 }
1072
1073 DPRINTF(("FB branch_location: %d\n", branch_location));
1074 set_feedback_branch(instance, branch_location);
1075
1076
1077
1078
1079
1080

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/initseq.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:43 pm | 10/82 |

```

1081  /* Find the next power of 2 number after sub_sequence */
1082  power_of_2_ceilng = 1;
1083  while (power_of_2_ceilng < sub_sequence) {
1084    power_of_2_ceilng <<= 1;
1085  }
1086
1087  dummy_count = 0;
1088
1089  if (power_of_2_ceilng != sub_sequence) {
1090    /* sub_sequence is not power of 2 long --> add more dummies:
1091     * 1. to make it power of 2 long
1092     * 2. to satisfy the BRANCH_LATENCY in the sub sequence.
1093     * Choose either 1 or 2 whichever produces less dummy ptrns.
1094     */
1095    dummy_count1 = power_of_2_ceilng - sub_sequence;
1096
1097    /* Note the following:
1098     * - branch_location is always odd
1099     * - BRANCH_LATENCY is even
1100     * - sub_sequence is even at this point
1101     * therefore dummy_count2 will always be even
1102     */
1103    dummy_count2 = (branch_location + 1) + BRANCH_LATENCY - sub_sequence;
1104    if (dummy_count2 < 0)
1105      dummy_count2 = 0;
1106
1107    if (dummy_count1 < dummy_count2)
1108      dummy_count = dummy_count1;
1109    else
1110      dummy_count = dummy_count2;
1111
1112    DPRINTF(("FB more dummy_count to satisfy branch constraint: %d\n",
1113      dummy_count));
1114
1115    for (i = 0; i < dummy_count; ++i) {
1116      ++fb_ptrn_count;
1117      write_pattern_unit(instance, dab_ptr->dummy_ptrn);
1118      if (grow_pattern_unit(instance) == FAILURE)
1119        return(FAILURE);
1120    }
1121  }
1122
1123  /* Replicate the feedback patterns to fill the whole feedback blocks */
1124  for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
1125
1126    if (dab_ptr->lane_used[lane0]) {
1127      dest_addr = instance->fb_block_addr[lane0] +
1128        fb_ptrn_count * PTRN_ADDR_INC;
1129
1130      /* Fill the first 512K patterns */
1131      number_of_copies = FB128K_FB_BLOCK_COUNT * PTRN_PER_BLOCK /
1132        fb_ptrn_count - 1;
1133      DPRINTF(("copy FB sequence %d times\n", number_of_copies));
1134      while (number_of_copies > 0) {
1135        copy_patterns(instance->fb_block_addr[lane0],
1136          dest_addr, fb_ptrn_count);
1137        dest_addr += fb_ptrn_count * PTRN_ADDR_INC;
1138        --number_of_copies;
1139      }
1140
1141      /* Fill the remaining blocks with dummy patterns */
1142      leftover = FB128K_FB_BLOCK_COUNT * PTRN_PER_BLOCK % fb_ptrn_count;
1143      DPRINTF(("FB fill block with dummy ptrns to 512 ptrns; count: %d\n",
1144        leftover));
1145
1146      dummy_long = (PTRN_BITS_LONGWORD * dab_ptr->dummy_ptrn[lane0]);
1147      while (leftover > 0) {
1148        write_loc_long((u_long *)dest_addr,
1149          dummy_long->word[0]);
1150        write_loc_long((u_long *)dest_addr + LANE_SEGMENT_B_OFFSET,
1151          dummy_long->word[1]);
1152        write_loc_long((u_long *)dest_addr + LANE_SEGMENT_C_OFFSET,
1153          dummy_long->word[2]);
1154
1155        DPRINTF(("FB dummy 408x: 408x 408x 408x\n", dest_addr,
1156          dummy_long->word[0],
1157          dummy_long->word[1],
1158          dummy_long->word[2]));
1159
1160        --leftover;
1161        dest_addr += PTRN_ADDR_INC;
1162      }
1163
1164      /* Replicate the first 512 patterns if necessary */
1165      number_of_copies =
1166        instance->fb_block_size[lane0] / FB128K_FB_BLOCK_COUNT - 1;
1167      DPRINTF(("FB replicate first 512 patterns for %d times\n",
1168        number_of_copies));
1169      while (number_of_copies > 0) {
1170        copy_patterns(instance->fb_block_addr[lane0], dest_addr,
1171          FB128K_FB_BLOCK_COUNT * PTRN_PER_BLOCK);
1172        dest_addr += FB128K_FB_BLOCK_COUNT * PTRN_PER_BLOCK *
1173          PTRN_ADDR_INC;
1174        --number_of_copies;
1175      }
1176
1177      /* Increment the pattern count to the maximum feedback block. Note
1178       * that this number is greater than the actual pattern memory used
1179       * if the number of feedback block required on each lane is different.
1180       */
1181      max_block_size = 0;
1182      for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
1183        if (instance->fb_block_size[lane0] > max_block_size)
1184          max_block_size = instance->fb_block_size[lane0];
1185      }
1186      instance->pattern_count += (max_block_size * PTRN_PER_BLOCK) -
1187        fb_ptrn_count;
1188      if (allocate_initial_block(instance) == FAILURE)
1189        return(FAILURE);
1190
1191      /* subtract pattern count which was incremented in
1192       * allocate_initial_block().
1193       */
1194      instance->pattern_count -= dab_ptr->unit_count_per_lane;
1195    }
1196  }

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/initseq.c | DATE 5/23/89 | PAGE # 11/83 |
|--|--|---|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 1201 | | /* Set the branch address for each block in the feedback sequence to | | |
| 1202 | | * link the post feedback blocks to the feedback blocks. | | |
| 1203 | | */ | | |
| 1204 | | for (lane = 0; lane < MAX_LANE_COUNT; ++lane) { | | |
| 1205 | | if (dab_ptr->lane_used[lane]) { | | |
| 1206 | | for (j = 0; j < instance->fb_block_count[lane]; ++j) { | | |
| 1207 | | write_branch_table(instance->fb_block_addr[lane] + | | |
| 1208 | | j * BLOCK_ADDR_INC, | | |
| 1209 | | instance->lane_addr[lane].max_addr - | | |
| 1210 | | BLOCK_ADDR_INC); | | |
| 1211 | | } | | |
| 1212 | | } | | |
| 1213 | | } | | |
| 1214 | | /* Now copy the part of the feedback sequence to the post feedback blocks | | |
| 1215 | | * because we might have played only part of the feedback sequence due | | |
| 1216 | | * to the BRANCH LATENCY. | | |
| 1217 | | */ | | |
| 1218 | | /* branch_location + 1 -> to get count instead of pattern number */ | | |
| 1219 | | played = (branch_location + 1 + BRANCH_LATENCY) % | | |
| 1220 | | (sub_sequence + dummy_count); | | |
| 1221 | | ptrs_to_copy = 0; | | |
| 1222 | | if (physical_seq_len > played) | | |
| 1223 | | ptrs_to_copy = physical_seq_len - played; | | |
| 1224 | | for (lane = 0; lane < MAX_LANE_COUNT; ++lane) { | | |
| 1225 | | if (dab_ptr->lane_used[lane]) { | | |
| 1226 | | source_addr[lane] = instance->fb_block_addr[lane] + | | |
| 1227 | | played * PTRN_ADDR_INC; | | |
| 1228 | | } | | |
| 1229 | | } | | |
| 1230 | | instance->pattern_count += ptrs_to_copy; | | |
| 1231 | | PRINTF(("copy fb sequence to post feedback blocks, ptrs_to_copy: %d\n", | | |
| 1232 | | ptrs_to_copy)); | | |
| 1233 | | /* ptrs_to_copy can be up to 512 patterns, so we cannot | | |
| 1234 | | * just do a copy_pattern() because we have only allocated 1 block. | | |
| 1235 | | */ | | |
| 1236 | | if (ptrs_to_copy > PTRN_PER_BLOCK) { | | |
| 1237 | | for (lane = 0; lane < MAX_LANE_COUNT; ++lane) { | | |
| 1238 | | if (dab_ptr->lane_used[lane]) { | | |
| 1239 | | copy_pattern(source_addr[lane], | | |
| 1240 | | instance->lane_addr[lane].max_addr - BLOCK_ADDR_INC, | | |
| 1241 | | PTRN_PER_BLOCK); | | |
| 1242 | | source_addr[lane] += BLOCK_ADDR_INC; | | |
| 1243 | | saved_block_addr[lane] = instance->lane_addr[lane].max_addr - | | |
| 1244 | | BLOCK_ADDR_INC; | | |
| 1245 | | } | | |
| 1246 | | } | | |
| 1247 | | ptrs_to_copy -= PTRN_PER_BLOCK; | | |
| 1248 | | if (allocate_initial_block(instance) == FAILURE) | | |
| 1249 | | return(FAILURE); | | |
| 1250 | | /* subtract pattern count which was incremented in | | |
| 1251 | | * allocate_initial_block(). | | |
| 1252 | | */ | | |
| 1253 | | instance->pattern_count -= dab_ptr->unit_count_per_lane; | | |
| 1254 | | for (lane = 0; lane < MAX_LANE_COUNT; ++lane) { | | |
| 1255 | | if (dab_ptr->lane_used[lane]) { | | |
| 1256 | | set_branch(saved_block_addr[lane] + BLOCK_ADDR_INC - | | |
| 1257 | | PTRN_ADDR_INC, | | |
| 1258 | | instance->lane_addr[lane].max_addr - BLOCK_ADDR_INC); | | |
| 1259 | | } | | |
| 1260 | | } | | |
| 1261 | | if (ptrs_to_copy > 0) { | | |
| 1262 | | for (lane = 0; lane < MAX_LANE_COUNT; ++lane) { | | |
| 1263 | | if (dab_ptr->lane_used[lane]) { | | |
| 1264 | | copy_pattern(source_addr[lane], | | |
| 1265 | | instance->lane_addr[lane].max_addr - BLOCK_ADDR_INC, | | |
| 1266 | | ptrs_to_copy); | | |
| 1267 | | } | | |
| 1268 | | } | | |
| 1269 | | /*overflowed_pattern_count = ptrs_to_copy; | | |
| 1270 | | /* Setup the pattern unit addresses. | | |
| 1271 | | * make instance->unit_addr[] points to the last pattern which has | | |
| 1272 | | * already been copied by copy_pattern(). When we do this it is possible | | |
| 1273 | | * that some instance->unit_addr[] will point to addresses less than the | | |
| 1274 | | * beginning address of the block. But this is OK because we will call | | |
| 1275 | | * grow_pattern() next which will fix this problem. grow_pattern() will | | |
| 1276 | | * make instance->unit_addr[] points to the address to load the next | | |
| 1277 | | * pattern. | | |
| 1278 | | */ | | |
| 1279 | | /* Calculate the addr_offset for each unit addr. Note that we can pick | | |
| 1280 | | * any unit in any lane to calculate the offset, since this offset is | | |
| 1281 | | * the same for any unit. Also note that the addr_offset could be negative | | |
| 1282 | | * if the ptrs_to_copy is less than unit_count_per_lane. | | |
| 1283 | | */ | | |
| 1284 | | lane = dab_ptr->unit_location(0).lane_no; | | |
| 1285 | | temp_unit_addr = instance->unit_addr[instance->cur_unit_addr_index[0]]; | | |
| 1286 | | addr_offset = instance->lane_addr[lane].max_addr - BLOCK_ADDR_INC + | | |
| 1287 | | ptrs_to_copy * PTRN_ADDR_INC - | | |
| 1288 | | dab_ptr->unit_count_per_lane * PTRN_ADDR_INC - | | |
| 1289 | | temp_unit_addr[0]; | | |
| 1290 | | total_unit = dab_ptr->lane_count * dab_ptr->unit_count_per_lane; | | |
| 1291 | | for (unitno = 0; unitno < total_unit; ++unitno) { | | |
| 1292 | | temp_unit_addr[unitno] += addr_offset; | | |
| 1293 | | } | | |
| 1294 | | if (grow_pattern(instance) == FAILURE) | | |
| 1295 | | return(FAILURE); | | |
| 1296 | | return(SUCCESS); | | |
| 1297 | | | | |
| 1298 | | | | |
| 1299 | | | | |
| 1300 | | | | |
| 1301 | | | | |
| 1302 | | | | |
| 1303 | | | | |
| 1304 | | | | |
| 1305 | | | | |
| 1306 | | | | |
| 1307 | | | | |
| 1308 | | | | |
| 1309 | | | | |
| 1310 | | | | |
| 1311 | | | | |
| 1312 | | | | |
| 1313 | | | | |
| 1314 | | | | |
| 1315 | | | | |
| 1316 | | | | |
| 1317 | | | | |
| 1318 | | | | |
| 1319 | | | | |
| 1320 | | | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/initseq.c

DATE

5/23/89

PAGE #

TIME 6:14:43 pm

12/84

```

1321
1322 load_post_feedback_seq(instance)
1323 INSTANCE_INFO *instance;
1324 {
1325     DEVICE_SPEC *def;
1326     SEQ_SPEC *seq_spec_ptr;
1327     UNB_OFFSET *unb_ptr;
1328     char *bit_array;
1329     u_short i;
1330     u_short pin_count;
1331     u_short word_offset;
1332     u_short bit_offset;
1333     u_short pin_number;
1334     u_short post_seq_length;
1335     u_short seq_count;
1336     u_char unitno;
1337     u_char wordno;
1338     u_char bitno;
1339
1340     DPRINTF(("inside load_post_feedback_seq\n"));
1341
1342     def = instance->definition;
1343     seq_count = def->seq_cnt;
1344     post_seq_length = def->post_seq_len;
1345
1346     for (i = 0; i < post_seq_length; ++i) {
1347
1348         word_offset = i / 8;
1349         bit_offset = 7 - i % 8;
1350
1351         for (pin_count = 0; pin_count < seq_count; ++pin_count) {
1352
1353             seq_spec_ptr = def->seq_table[pin_count];
1354             bit_array = seq_spec_ptr->post_bits;
1355             pin_number = seq_spec_ptr->pin_number;
1356
1357             unb_ptr = seq_to_short_offset(pin_number);
1358             unitno = unb_ptr->unitno;
1359             wordno = unb_ptr->wordno;
1360             bitno = unb_ptr->bitno;
1361
1362             if ((bit_array[word_offset] >> bit_offset) & 1) {
1363                 set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
1364             }
1365             else {
1366                 reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
1367             }
1368         }
1369
1370         write_pattern(instance,
1371                     instance->unit_addr[instance->cur_unit_addr_index][0],
1372                     instance->ptrn_loaded);
1373         if (grow_pattern(instance) == FAILURE)
1374             return(FAILURE);
1375     }
1376     return(SUCCESS);
1377
1378
1379 set_initial_values(instance)
1380 INSTANCE_INFO *instance;
1381 {
1382     DEVICE_SPEC *def_ptr;
1383     EXTRA_DEVICE_SPEC *extra_def_ptr;
1384     DAB_INFO *dab_ptr;
1385     PIN_SPEC *pin_spec_ptr;
1386     PIN_INFO *pin_info;
1387     UNB_OFFSET *unb_ptr;
1388     PTRN_BITS *ptrn_bits_ptr;
1389     PTRN_BITS_LONGWORD *ptrn_loaded_ptr;
1390     PTRN_BITS_LONGWORD *last_consistent_set_ptr;
1391     PTRN_BITS_LONGWORD *ident_r1_ptr;
1392     PTRN_BITS_LONGWORD *ident_store_ptr;
1393     PTRN_BITS_LONGWORD *ident_outputs_ptr;
1394     PTRN_BITS_LONGWORD *ident_R1_ptr;
1395     PTRN_BITS_LONGWORD *ident_R2_ptr;
1396     PTRN_BITS_LONGWORD *hndcnb_loaded_ptr;
1397     PTRN_BITS_LONGWORD *last_sample_value_hiz_ptr;
1398     u_long last_block_number[MAX_LANE_COUNT];
1399     u_long ident_store_loc;
1400     u_long seq_end_addr[MAX_LANE_COUNT];
1401     u_short seq_count;
1402     u_short pin_count;
1403     u_short pin_number;
1404     u_char total_unit;
1405     u_char unitno;
1406     u_char wordno;
1407     u_char bitno;
1408     u_char changed_dac;
1409
1410     DPRINTF(("set initial values\n"));
1411
1412     def_ptr = instance->definition;
1413     extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
1414
1415     dab_ptr = dab_list[instance->dab_info_index];
1416     total_unit = dab_ptr->unit_count;
1417
1418     ptrn_loaded_ptr = (PTRN_BITS_LONGWORD *)instance->ptrn_loaded;
1419     ident_R1_ptr = (PTRN_BITS_LONGWORD *)extra_def_ptr->ident_R1;
1420     ident_R2_ptr = (PTRN_BITS_LONGWORD *)extra_def_ptr->ident_R2;
1421     ident_outputs_ptr = (PTRN_BITS_LONGWORD *)extra_def_ptr->ident_outputs;
1422     for (unitno = 0; unitno < total_unit; ++unitno) {
1423         for (wordno = 0; wordno < 3; ++wordno) {
1424
1425             /* Disable the R1 and R2 Clock */
1426             ptrn_loaded_ptr->word[wordno] |= ident_R1_ptr->word[wordno];
1427             ptrn_loaded_ptr->word[wordno] |= ident_R2_ptr->word[wordno];
1428
1429             ptrn_loaded_ptr->word[wordno] |= ident_outputs_ptr->word[wordno];
1430         }
1431
1432         ++ptrn_loaded_ptr;
1433         ++ident_R1_ptr;
1434         ++ident_R2_ptr;
1435         ++ident_outputs_ptr;
1436     }
1437
1438     write_pattern(instance,
1439                 instance->unit_addr[instance->cur_unit_addr_index][0],
1440                 instance->ptrn_loaded);

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/initseq.c | DATE 5/23/89 | PAGE # 13/85 |
|--|--|---|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 1441 | | endifdef DBASE | | |
| 1442 | | if (start_seq(def_ptr) == FAILURE) { | | |
| 1443 | | return(FAILURE); | | |
| 1444 | | } | | |
| 1445 | | endif | | |
| 1446 | | if (program_dac(def_ptr, (char)instance->dab_info_index, | | |
| 1447 | | changed_dac) == FAILURE) { | | |
| 1448 | | return(FAILURE); | | |
| 1449 | | } | | |
| 1450 | | if (set_edge_and_sample_setting(extra_def_ptr, | | |
| 1451 | | (u_long)RESOLUTION_05_NS, | | |
| 1452 | | (u_long)MAX_SAMPLE_RANGE_RANGE) == FAILURE) | | |
| 1453 | | return(FAILURE); | | |
| 1454 | | set_seq_end_bit(instance, | | |
| 1455 | | instance->last_addr, | | |
| 1456 | | FALSE, | | |
| 1457 | | seq_end_addr, | | |
| 1458 | | last_block_number); | | |
| 1459 | | if (play_ptr_seq(instance, (u_long)PTEN_PLAY_TIMEOUT, | | |
| 1460 | | changed_dac) == FAILURE) | | |
| 1461 | | return(FAILURE); | | |
| 1462 | | endifdef DEBUG | | |
| 1463 | | if (loop_till_key == TRUE) { | | |
| 1464 | | printf("looping in create instance\n"); | | |
| 1465 | | debug_key = 0; | | |
| 1466 | | while (debug_key == 0) { | | |
| 1467 | | if (play_ptr_seq(instance, | | |
| 1468 | | (u_long)PTEN_PLAY_TIMEOUT, changed_dac) == FAILURE) | | |
| 1469 | | return(FAILURE); | | |
| 1470 | | } | | |
| 1471 | | if (debug_char == (u_long)'i') { | | |
| 1472 | | loop_till_key = FALSE; | | |
| 1473 | | printf("stop loop\n"); | | |
| 1474 | | } | | |
| 1475 | | debug_key = 0; | | |
| 1476 | | } | | |
| 1477 | | endif | | |
| 1478 | | remove_seq_end_bit(instance, seq_end_addr, last_block_number); | | |
| 1479 | | read_magic_full_sample_reg(instance, instance->last_sample_value); | | |
| 1480 | | /* Set the initial simulator pin values to whatever value we sample. */ | | |
| 1481 | | copy_ptr_bits((char *)instance->last_sample_value.unknown, | | |
| 1482 | | (char *)instance->sim_pin_value.unknown, | | |
| 1483 | | dab_ptr->unit_count); | | |
| 1484 | | copy_ptr_bits((char *)instance->last_sample_value.hiz, | | |
| 1485 | | (char *)instance->sim_pin_value.hiz, | | |
| 1486 | | dab_ptr->unit_count); | | |
| 1487 | | copy_ptr_bits((char *)instance->last_sample_value.data, | | |
| 1488 | | (char *)instance->sim_pin_value.data, | | |
| 1489 | | dab_ptr->unit_count); | | |
| 1490 | | /* Set the sim_pin_value to the last value of the reset sequence. | | |
| 1491 | | * Note that this value is in the last_ptrn loaded. | | |
| 1492 | | * Also note that for R1/R2 pins, set the sim_pin_value to the inactive | | |
| 1493 | | * state. | | |
| 1494 | | */ | | |
| 1495 | | seq_count = def_ptr->seq_cnt; | | |
| 1496 | | for (pin_count = 0; pin_count < seq_count; ++pin_count) { | | |
| 1497 | | pin_number = def_ptr->seq_table[pin_count].pin_number; | | |
| 1498 | | pin_spec_ptr = iddef_ptr->pin_table[pin_number]; | | |
| 1499 | | pin_info = instance->pin_info_table[pin_number]; | | |
| 1500 | | word_ptr = &pin_info->word_offset[pin_number]; | | |
| 1501 | | unitno = word_ptr->unitno; | | |
| 1502 | | wordno = word_ptr->wordno; | | |
| 1503 | | bitno = word_ptr->bitno; | | |
| 1504 | | switch (pin_spec_ptr->clk_format) { | | |
| 1505 | | case DWR1: | | |
| 1506 | | pin_info->uninitialized_pin = FALSE; | | |
| 1507 | | if (read_ptrn_bit(instance->ptrn_loaded(unitno), wordno, bitno) == 0) | | |
| 1508 | | set_pin_value(instance->sim_pin_value, | | |
| 1509 | | unitno, wordno, bitno, (u_char)LOGIC_0); | | |
| 1510 | | else | | |
| 1511 | | set_pin_value(instance->sim_pin_value, | | |
| 1512 | | unitno, wordno, bitno, (u_char)LOGIC_1); | | |
| 1513 | | break; | | |
| 1514 | | case R1: | | |
| 1515 | | pin_info->uninitialized_pin = FALSE; | | |
| 1516 | | set_pin_value(instance->sim_pin_value, | | |
| 1517 | | unitno, wordno, bitno, (u_char)LOGIC_1); | | |
| 1518 | | break; | | |
| 1519 | | case R0: | | |
| 1520 | | pin_info->uninitialized_pin = FALSE; | | |
| 1521 | | set_pin_value(instance->sim_pin_value, | | |
| 1522 | | unitno, wordno, bitno, (u_char)LOGIC_0); | | |
| 1523 | | break; | | |
| 1524 | | default: | | |
| 1525 | | break; | | |
| 1526 | | } | | |
| 1527 | | /* Set the initial simulator pin value for I/O store pins to 1. | | |
| 1528 | | * Also set the hndcnb for I/O store pins which are driving to 1. | | |
| 1529 | | */ | | |
| 1530 | | ident_ios_ptr = (PTEN_BITS_LONGWORD *)extra_def_ptr->ident_ios; | | |
| 1531 | | ident_store_ptr = (PTEN_BITS_LONGWORD *)extra_def_ptr->ident_store; | | |
| 1532 | | hndcnb_loaded_ptr = (PTEN_BITS_LONGWORD *)instance->hndcnb_loaded; | | |
| 1533 | | last_sample_value_hiz_ptr = | | |
| 1534 | | (PTEN_BITS_LONGWORD *)instance->last_sample_value.hiz; | | |
| 1535 | | for (unitno = 0; unitno < total_unit; ++unitno) { | | |
| 1536 | | for (wordno = 0; wordno < 1; ++wordno) { | | |
| 1537 | | ident_store_ios = ident_ios_ptr->word[wordno] & | | |
| 1538 | | ident_store_ptr->word[wordno]; | | |
| 1539 | | hndcnb_loaded_ptr->word[wordno] = | | |
| 1540 | | ident_store_ios & last_sample_value_hiz_ptr->word[wordno]; | | |
| 1541 | | } | | |
| 1542 | | } | | |
| 1543 | | ++ident_ios_ptr; | | |
| 1544 | | } | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/initseq.c

DATE 5/23/89
TIME 6:14:43 pm

PAGE #
14/86

```

1561  ++ident_store_ptr,
1562  ++hmdenb_loaded_ptr,
1563  ++last_sample_value_ptr,
1564  }
1565
1566  /* Calculate consistent set */
1567  calculate_consistent_set(instance, def_ptr, instance->last_consistent_set);
1568
1569  /* Restore the HMDENB to the actual value. The first HMDENB (in preamble)
1570   * disable the I/O driver for IO and OUTPUT pins.
1571   */
1572  write_pattern(instance,
1573               instance->unit_addr(instance->cur_unit_addr_index)[0],
1574               instance->hmdenb_loaded);
1575
1576  if (grow_pattern(instance) == FAILURE)
1577      return(FAILURE);
1578
1579  last_consistent_set_ptr = (PTRN_BITS_LONGWORD *)
1580                          instance->last_consistent_set;
1581  ptrn_bits_ptr = instance->last_consistent_set;
1582  ident_R1_ptr = (PTRN_BITS_LONGWORD *)extra_def_ptr->ident_R1;
1583  ident_R2_ptr = (PTRN_BITS_LONGWORD *)extra_def_ptr->ident_R2;
1584  for (unitno = 0; unitno < total_unit; ++unitno) {
1585      set_pel_user_bit(ptrn_bits_ptr->ctl);
1586
1587      for (wordno = 0; wordno < 3; ++wordno) {
1588
1589          /* Disable the R1 and R2 clock */
1590          last_consistent_set_ptr->word[wordno] |= ident_R1_ptr->word[wordno];
1591          last_consistent_set_ptr->word[wordno] |= ident_R2_ptr->word[wordno];
1592      }
1593
1594      ++last_consistent_set_ptr,
1595      ++ident_R1_ptr,
1596      ++ident_R2_ptr,
1597      ++ptrn_bits_ptr,
1598  }
1599
1600  /* Mail down the first pattern */
1601  write_pattern(instance,
1602               instance->unit_addr(instance->cur_unit_addr_index)[0],
1603               instance->last_consistent_set);
1604
1605  /* User bit in first pattern only. */
1606  ptrn_bits_ptr = instance->last_consistent_set;
1607  for (unitno = 0; unitno < total_unit; ++unitno) {
1608      clear_pel_user_bit(ptrn_bits_ptr->ctl);
1609      ++ptrn_bits_ptr;
1610  }
1611
1612  if (grow_pattern(instance) == FAILURE)
1613      return(FAILURE);
1614
1615  /* Duplicate the first pattern (with the trigger bit) because,
1616   * it can be overwritten by the HMDENB if any I/O store pins
1617   * change direction from driving to Z.
1618   */
1619  write_pattern(instance,
1620               instance->unit_addr(instance->cur_unit_addr_index)[0],
1621               instance->last_consistent_set);
1622
1623  if (grow_pattern(instance) == FAILURE)
1624      return(FAILURE);
1625
1626  /* copy the last_consistent_set to ptrn_loaded */
1627  copy_ptrn_bits((char *)instance->last_consistent_set,
1628               (char *)instance->ptrn_loaded,
1629               dab_ptr->unit_count);
1630
1631  /* Initialize the OUTPUT pins in ptrn_loaded to 0 */
1632  ptrn_loaded_ptr = (PTRN_BITS_LONGWORD *)instance->ptrn_loaded;
1633  ident_outputs_ptr = (PTRN_BITS_LONGWORD *)extra_def_ptr->ident_outputs;
1634  for (unitno = 0; unitno < total_unit; ++unitno) {
1635      for (wordno = 0; wordno < 3; ++wordno) {
1636          ptrn_loaded_ptr->word[wordno] |= ident_outputs_ptr->word[wordno];
1637      }
1638      ++ptrn_loaded_ptr,
1639      ++ident_outputs_ptr,
1640  }
1641
1642  /* Write out the measurement pattern */
1643  write_pattern(instance,
1644               instance->unit_addr(instance->cur_unit_addr_index)[0],
1645               instance->ptrn_loaded);
1646
1647  return(SUCCESS);
1648
1649  }

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/lmserver.c | DATE 5/23/89 | PAGE # 1/87 |
|--|--|-------------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCCS ID: lmserver.c rev 1.2, 5/9/89 at 17:20:35 */ | | | |
| 2 | #include "device.h" | | | |
| 3 | #include "message.h" | | | |
| 4 | #include "re.h" | | | |
| 5 | #include "soc_x.h" | | | |
| 6 | #include "septom.h" | | | |
| 7 | #include "lmserver.h" | | | |
| 8 | #include "network.h" | | | |
| 9 | #include "function.h" | | | |
| 10 | #include "lm_sfi.h" | | | |
| 11 | #endif | | | |
| 12 | #ifdef MODELER | | | |
| 13 | #include "vrtx.h" | | | |
| 14 | #endif | | | |
| 15 | #ifdef DEBUG | | | |
| 16 | extern u_long debug_key; | | | |
| 17 | extern u_long debug_char; | | | |
| 18 | u_char loop_till_key = FALSE; | | | |
| 19 | #endif | | | |
| 20 | /* ??? defined in hvaran.h */ | | | |
| 21 | #define SHUTDOWN (char)2 | | | |
| 22 | #define REBOOT (char)3 | | | |
| 23 | #define TICKS_PER_SECOND 1000 | | | |
| 24 | #define REBOOT_MSG_DELAY_TIME (TICKS_PER_SECOND * 30) | | | |
| 25 | #endif | | | |
| 26 | #ifdef DIRECTCONN | | | |
| 27 | ec_spend(X, Y, Z) {} | | | |
| 28 | #endif | | | |
| 29 | u_char play_completed_flag; | | | |
| 30 | u_long available_malloc_size; | | | |
| 31 | u_long total_malloc_size; | | | |
| 32 | int play_semaphore; | | | |
| 33 | #endif | | | |
| 34 | #ifdef DEBUG | | | |
| 35 | u_char lm_pristit = TRUE; | | | |
| 36 | #endif | | | |
| 37 | CONNECTION *lm_global_conn_ptr; | | | |
| 38 | extern CONNECTION table_of_conn[1]; | | | |
| 39 | #endif | | | |
| 40 | DAB_INFO *dab_list[MAX_LANE_COUNT * MAX_SLOT_COUNT]; | | | |
| 41 | USER_INFO *user_info_array[MAX_USER_COUNT]; | | | |
| 42 | SYSTEM_INFO *system_config; | | | |
| 43 | #endif | | | |
| 44 | void (*function_array[MAX_FUNCTION])(); | | | |
| 45 | #endif | | | |
| 46 | CONFIGURATION_ERRORS config_error; | | | |
| 47 | #endif | | | |
| 48 | /* CPU byte address of free block on each lane */ | | | |
| 49 | u_long free_block_list[MAX_LANE_COUNT]; | | | |
| 50 | #endif | | | |
| 51 | u_long bitso_to_mask[32]; | | | |
| 52 | #endif | | | |
| 53 | /* ??? for debugging only */ | | | |
| 54 | #ifdef DBASE | | | |
| 55 | TM_RESULT tm_result_array[MAX_PIN_COUNT]; | | | |
| 56 | #endif | | | |
| 57 | u_char is_measure_delay; | | | |
| 58 | #endif | | | |
| 59 | PTM_BITS *gbl_dummy_ptr; | | | |
| 60 | #endif | | | |
| 61 | FULL_VALUE gbl_new_sample_value; | | | |
| 62 | FULL_VALUE gbl_steady_state_result; | | | |
| 63 | FULL_VALUE gbl_tamp_steady_state_result; | | | |
| 64 | PTM_BITS_LONGWORD gbl_ident_inconsistent_pins[MAX_UNIT_COUNT]; | | | |
| 65 | PTM_BITS_LONGWORD gbl_ident_change[MAX_UNIT_COUNT]; | | | |
| 66 | TM_PIN_INFO gbl_tm_pin_info_table[MAX_PIN_COUNT]; | | | |
| 67 | #endif | | | |
| 68 | /* The convert bit patterns to logic value. | | | |
| 69 | /* The syntax of the bit patterns is ONE:N12:SOFT:DATA | | | |
| 70 | */ | | | |
| 71 | u_char bit_to_logic_table[16]; | | | |
| 72 | #endif | | | |
| 73 | #ifdef MODELER | | | |
| 74 | LM_HARDWARE_ERROR modeler_error; | | | |
| 75 | #endif | | | |
| 76 | extern LM_HARDWARE_ERROR modeler_error; | | | |
| 77 | u_char fatal_hardware_error_encountered = FALSE; | | | |
| 78 | u_char fatal_configuration_error_encountered = FALSE; | | | |
| 79 | u_char non_fatal_configuration_error_encountered = FALSE; | | | |
| 80 | long time_reboot_was_issued; | | | |
| 81 | long reboot_delay_time; | | | |
| 82 | u_char rebooting = FALSE; | | | |
| 83 | u_char reboot_flag; | | | |
| 84 | u_char xabutdown = FALSE; | | | |
| 85 | #endif | | | |
| 86 | UWB_OFFSET ps_to_short_offset[MAX_PIN_COUNT]; | | | |
| 87 | #endif | | | |
| 88 | u_short gbl_eval_pin_number[MAX_EVAL_CHANGES]; | | | |
| 89 | u_char gbl_eval_pin_value[MAX_EVAL_CHANGES]; | | | |
| 90 | u_long gbl_eval_pin_delay[MAX_EVAL_CHANGES]; | | | |
| 91 | u_long gbl_eval_max_delay[MAX_EVAL_CHANGES]; | | | |
| 92 | #endif | | | |
| 93 | u_char lm_hardware_init_done = FALSE; | | | |
| 94 | #endif | | | |
| 95 | extern u_long lm_dead_user_mask; | | | |
| 96 | #endif | | | |
| 97 | #ifdef MODELER | | | |
| 98 | main() | | | |
| 99 | { | | | |
| 100 | void | | | |
| 101 | pattern_task() | | | |
| 102 | #endif | | | |
| 103 | { | | | |
| 104 | u_short rtn; | | | |
| 105 | u_char user; | | | |
| 106 | u_short type; | | | |
| 107 | u_long temp; | | | |
| 108 | u_long mins, secs; | | | |
| 109 | char str[MAX_MESSAGE]; | | | |
| 110 | u_long *reboot_msg_delay = (u_long*) NULL; | | | |
| 111 | #endif | | | |
| 112 | { | | | |
| 113 | u_short rtn; | | | |
| 114 | u_char user; | | | |
| 115 | u_short type; | | | |
| 116 | u_long temp; | | | |
| 117 | u_long mins, secs; | | | |
| 118 | char str[MAX_MESSAGE]; | | | |
| 119 | u_long *reboot_msg_delay = (u_long*) NULL; | | | |
| 120 | #endif | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/lmserver.c

DATE 5/23/89
TIME 6:14:45 pm

PAGE #
2/88

```

LINE # SOURCE TEXT
121 extern u_short lm_give_me_the_next_user();
122
123 DPRINTF(("lm server\n"));
124
125 #ifdef MODELER
126 init();
127
128 if (init_socket() == FAILURE) {
129     DPRINTF(("ERROR in init_socket\n"));
130 }
131
132 read_hw_config();
133
134 init_mod_err();
135
136 (void)printf("ready\n");
137 #endif
138
139 lm_dead_user_mask = 0;
140 while (1) {
141     rta = lm_give_me_the_next_user(user);
142     DPRINTF(("got user: id from lm_give_me_the_next_user\n", user));
143
144     lm_global_conn_ptr = table_of_conns[user];
145     if (lm_global_conn_ptr == NULL)
146         continue;
147
148     if (rta != SUCCESS) {
149         while (lm_dequeue_message(&type, &str) == SUCCESS) {
150             DPRINTF(("ta\n", str));
151         }
152         DPRINTF(("error in lm_give_me_the_next_user()\n"));
153     }
154     else {
155         lm_flush_message_queue();
156
157         if (lm_hardware_init_done == FALSE) {
158             send_hardware_init_message();
159             continue;
160         }
161
162         if (user_info_array[user] != active == FALSE) {
163             if (rebooting == FALSE) {
164                 if (prepare_user(user) == FAILURE) {
165                     send_alloc_error_message(user_info_array[user]);
166                     check_reboot();
167                     continue;
168                 }
169             }
170
171             if (non_fatal_configuration_error_encountered == TRUE) {
172                 queue_up_non_fatal_config_message();
173             }
174
175             process_client_request(user_info_array[user]);
176
177             else {
178                 if (prepare_user(user) == FAILURE) {
179                     send_alloc_error_message(user_info_array[user]);
180                     check_reboot();
181                     continue;
182                 }
183
184                 send_rebooting_message(user_info_array[user]);
185             }
186         }
187         else {
188             if (rebooting == TRUE) {
189                 if (reboot_msg_delay == (u_long *) NULL) {
190                     reboot_msg_delay = (u_long *) DCALLOC(MAX_USER_COUNT, sizeof(u_long));
191                 }
192                 tmp = lm_time() - time_reboot_was_issued;
193                 if (tmp > (reboot_msg_delay + user)) {
194                     if (reboot_delay_time > tmp) {
195                         mins = (reboot_delay_time - tmp) / (60 * TICKS_PER_SECOND);
196                         secs = ((reboot_delay_time - tmp) / TICKS_PER_SECOND) - (mins * 60);
197                         lm_queue_message(WARNING_MSG, "Modeler is going down in %d minutes and %d seconds.",
198                                         mins, secs);
199                     }
200                     else {
201                         lm_queue_message(WARNING_MSG, "Modeler is going down when all users are finished.");
202                     }
203                     *(reboot_msg_delay + user) = tmp + REBOOT_MSG_DELAY_TIME;
204                 }
205             }
206             process_client_request(user_info_array[user]);
207         }
208     }
209     check_reboot();
210 }
211
212 #endif
213
214 process_client_request(cur_user)
215 USER_INFO *cur_user;
216 {
217     u_short errors;
218     u_short warnings;
219     char func_number;
220
221     DPRINTF(("inside process_client_request\n"));
222
223     #ifdef DEBUG
224     if (debug_key == 1) {
225         printf("key hit: %x\n", debug_char);
226         if (debug_char == (u_long)'p') {
227             if (lm_printit == TRUE)
228                 lm_printit = FALSE;
229             else
230                 lm_printit = TRUE;
231         }
232         else if ((debug_char == (u_long)'t') {
233             if (loop_till_key == TRUE)
234                 loop_till_key = FALSE;
235             else
236                 loop_till_key = TRUE;
237         }
238     }
239 }
240

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/lmsrver.c

DATE 5/23/89
TIME 6:14:45 pm

PAGE #
3/89

```

LINE # SOURCE TEXT
241     else {
242         printf("unknown key hit\n");
243     }
244     debug_key = 0;
245 }
246 #endif
247
248 if (fatal_hardware_error_encountered == TRUE) {
249     DPRINTF(("fatal_hardware_error_encountered == TRUE\n"));
250     send_fatal_hw_error_message(cur_user);
251     return;
252 }
253 else if (fatal_configuration_error_encountered == TRUE) {
254     DPRINTF(("fatal_configuration_error_encountered == TRUE\n"));
255     send_fatal_config_error_message(cur_user);
256     return;
257 }
258 else {
259     if (modeler_error.error == TRUE) {
260         DPRINTF(("modeler_error.error == TRUE\n"));
261         modeler_error.error = FALSE;
262     }
263     if ((modeler_error.pac_lane_errors != 0) ||
264         (modeler_error.lms_error == TRUE) ||
265         (modeler_error.unknown_source_of_interrupt == TRUE)) {
266         fatal_hardware_error_encountered = TRUE;
267         send_fatal_hw_error_message(cur_user);
268         return;
269     }
270     if (modeler_error.dab_change) {
271         reconfigure_dab();
272     }
273     else {
274         if (modeler_error.pel_error_list != 0) {
275             /* Set it back to TRUE so that play_ptrseq() is aware
276              * of this condition.
277             */
278             modeler_error.error = TRUE;
279         }
280     }
281 }
282
283 func_number = (lm_get_int() - 2) >> 1;
284
285 if ((func_number >= 0) && (func_number < MAX_FUNCTION)) {
286     function_array[func_number](cur_user);
287 }
288 else
289     process_no_such(cur_user);
290
291 lm_message_types(&errors, &warnings);
292 if ((errors + warnings) != 0)
293     DPRINTF(("ASSERT: some messages not dequeued at end of process_\n"));
294
295 #endif
296
297 send_hardware_init_message()
298 {
299     char command_number;
300 #ifdef DEBUG
301     u_short temp;
302     char error_string[MAX_STRING_LENGTH];
303 #endif
304 #endif
305
306     DPRINTF(("inside send_hardware_init_message\n"));
307
308     command_number = lm_get_int();
309
310     lm_put_int(command_number + 1);
311     lm_queue_message(ERROR_MSG, "modeler is being initialized");
312     end_queue_message();
313
314     end_put(0);
315
316     lm_delay(50);
317
318     if (close_connection_for_server(lm_global_conn_ptr) != SUCCESS) {
319 #ifdef DEBUG
320         while (1) {
321             if (lm_dequeue_message(&temp, error_string) != FAILURE)
322                 DPRINTF(("ls", error_string));
323             else
324                 break;
325         }
326 #endif
327     }
328 }
329
330
331 send_rebooting_message(user)
332 USER_INFO *user;
333 {
334 #ifdef DEBUG
335     u_short temp;
336     char error_string[512];
337 #endif
338 #endif
339     char command_number;
340
341     DPRINTF(("inside send_rebooting_message\n"));
342
343     command_number = lm_get_int();
344
345     reset_obuf();
346     lm_put_int(command_number + 1);
347     if (xshutdown == FALSE) {
348         lm_queue_message(ERROR_MSG, "cannot accept any new users; modeler is being rebooted...");
349     } else {
350         lm_queue_message(ERROR_MSG, "cannot accept any new users; modeler is being shut down...");
351     }
352
353     abort_user(user);
354
355     end_queue_message();
356
357     end_put(user->fd);
358
359     if (set_close_connection_for_server(lm_global_conn_ptr) != SUCCESS) {
360 #ifdef DEBUG

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/lmserver.c

DATE 5/23/89
TIME 6:14:45 pm

PAGE #
4/90

| LINE # | SOURCE TEXT |
|--------|--|
| 361 | while (1) { |
| 362 | if (lm_dequeue_message(&temp, error_string) != FAILURE) |
| 363 | DPRINTF(("ls", error_string)); |
| 364 | else |
| 365 | break; |
| 366 | } |
| 367 | endif |
| 368 | } |
| 369 | } |
| 370 | |
| 371 | send_fatal_hw_error_message(user) |
| 372 | USER_INFO "user," |
| 373 | { |
| 374 | #ifdef DEBUG |
| 375 | u_short temp; |
| 376 | char error_string[512]; |
| 377 | endif |
| 378 | char command_number; |
| 379 | |
| 380 | DPRINTF(("inside send_fatal_hw_error_message\n")); |
| 381 | |
| 382 | command_number = lm_get_int(); |
| 383 | |
| 384 | reset_obuf(); |
| 385 | lm_put_int(command_number + 1); |
| 386 | |
| 387 | get_fatal_hardware_message(); |
| 388 | |
| 389 | abort_user(user); |
| 390 | |
| 391 | end_queue_message(); |
| 392 | |
| 393 | end_put(user->fd); |
| 394 | |
| 395 | |
| 396 | if (set_close_connection_for_server(lm_global_conn_ptr) != SUCCESS) { |
| 397 | #ifdef DEBUG |
| 398 | while (1) { |
| 399 | if (lm_dequeue_message(&temp, error_string) != FAILURE) |
| 400 | DPRINTF(("ls", error_string)); |
| 401 | else |
| 402 | break; |
| 403 | } |
| 404 | endif |
| 405 | } |
| 406 | } |
| 407 | |
| 408 | send_fatal_config_error_message(user) |
| 409 | USER_INFO "user," |
| 410 | { |
| 411 | u_short temp; |
| 412 | char command_number; |
| 413 | #ifdef DEBUG |
| 414 | char error_string[512]; |
| 415 | endif |
| 416 | |
| 417 | DPRINTF(("inside send_fatal_config_error_message\n")); |
| 418 | |
| 419 | command_number = lm_get_int(); |
| 420 | |
| 421 | reset_obuf(); |
| 422 | lm_put_int(command_number + 1); |
| 423 | |
| 424 | lm_queue_message(ERROR_MSG, "fatal error encountered during hardware configuration"); |
| 425 | if (config_error.no_tmg) { |
| 426 | lm_queue_message(ERROR_MSG, "Timing Generator is offline"); |
| 427 | } |
| 428 | |
| 429 | if (config_error.tmg_cal) { |
| 430 | lm_queue_message(ERROR_MSG, "Timing Generator error: calibration failed"); |
| 431 | } |
| 432 | |
| 433 | if (config_error.no_pam_for_pac) { |
| 434 | for (temp = 0; temp < MAX_LANE_COUNT; ++temp) { |
| 435 | if (config_error.no_pam_for_pac & (1 << temp)) { |
| 436 | lm_queue_message(ERROR_MSG, "no Fast Pattern Memory in lane: %c", |
| 437 | 'A' + temp); |
| 438 | } |
| 439 | } |
| 440 | |
| 441 | |
| 442 | if (config_error.pam_strapped_wrong) { |
| 443 | for (temp = 0; temp < MAX_LANE_COUNT * MAX_PAM_COUNT; ++temp) { |
| 444 | if (config_error.pam_strapped_wrong & (1 << temp)) { |
| 445 | lm_queue_message(ERROR_MSG, "Fast Pattern Memory: %d in lane: %c is strapped incorrectly", |
| 446 | temp % MAX_PAM_COUNT, |
| 447 | 'A' + temp / MAX_PAM_COUNT); |
| 448 | } |
| 449 | } |
| 450 | |
| 451 | |
| 452 | if (config_error.pam_stacked_wrong) { |
| 453 | for (temp = 0; temp < MAX_LANE_COUNT * MAX_PAM_COUNT; ++temp) { |
| 454 | if (config_error.pam_stacked_wrong & (1 << temp)) { |
| 455 | lm_queue_message(ERROR_MSG, "Fast Pattern Memory: %d in lane: %c is stacked incorrectly", |
| 456 | temp % MAX_PAM_COUNT, |
| 457 | 'A' + temp / MAX_PAM_COUNT); |
| 458 | } |
| 459 | } |
| 460 | |
| 461 | |
| 462 | abort_user(user); |
| 463 | |
| 464 | end_queue_message(); |
| 465 | |
| 466 | end_put(user->fd); |
| 467 | |
| 468 | |
| 469 | if (set_close_connection_for_server(lm_global_conn_ptr) != SUCCESS) { |
| 470 | #ifdef DEBUG |
| 471 | while (1) { |
| 472 | if (lm_dequeue_message(&temp, error_string) != FAILURE) |
| 473 | DPRINTF(("ls", error_string)); |
| 474 | else |
| 475 | break; |
| 476 | } |
| 477 | endif |
| 478 | } |
| 479 | } |
| 480 | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/lmserver.c

DATE

5/23/89

PAGE #

TIME

6:14:45 pm

5/91

```

LINE # SOURCE TEXT
481 read_alloc_error_message(user)
482 USER_INFO *user;
483 {
484 #ifdef DEBUG
485     u_short temp;
486     char error_string[512];
487 #endif
488     char command_number;
489
490     DPRINTF(("inside read_alloc_error_message\n"));
491
492     command_number = lm_get_int();
493
494     reset_cbuf();
495     lm_put_int(command_number + 1);
496     lm_queue_message(ERROR_MSG, "out of memory on modeler; user aborted...");
497     abort_user(user);
498     end_queue_message();
499     end_put(user->fd);
500
501     if (set_close_connection_for_server(lm_global_conn_ptr) != SUCCESS) {
502 #ifdef DEBUG
503         while (1) {
504             if (lm_dequeue_message(&temp, error_string) != FAILURE)
505                 DPRINTF(("lm", error_string));
506             else
507                 break;
508         }
509 #endif
510     }
511     check_reboot();
512     /* Read any modeler address space to force it to interrupt if there is
513     * any dab configuration changes.
514     */
515     (void)probe((u_long)CLOS_START_ADDR);
516
517     if (rebooting == TRUE) {
518         if ((lm_time() - time_reboot_was_issued) > reboot_delay_time) {
519             /* The number of seconds we need to wait has elapsed */
520             if (reboot_flag == LM_FALSE) {
521                 /* We don't need to wait for all users to finish -->
522                 * just hit the software reset button now.
523                 */
524                 if (shutdown == FALSE) {
525 #ifdef MODELER
526                     lm_delay(1000);
527                     reset_cpu(REBOOT);
528 #endif
529                 }
530                 else {
531 #ifdef MODELER
532                     lm_delay(1000);
533                     reset_cpu(SHUTDOWN);
534 #endif
535                 }
536             }
537             else {
538                 if (any_active_user() == FALSE) {
539                     if (shutdown == FALSE) {
540 #ifdef MODELER
541                     lm_delay(1000);
542                     reset_cpu(REBOOT);
543 #endif
544                 }
545                 else {
546 #ifdef MODELER
547                     lm_delay(1000);
548                     reset_cpu(SHUTDOWN);
549 #endif
550                 }
551             }
552         }
553     }
554
555     any_active_user()
556     {
557         u_char i;
558
559         for (i = 0; i < MAX_USER_COUNT; ++i)
560             if (user_info_array[i]->active == TRUE)
561                 return(TRUE);
562         return(FALSE);
563     }
564
565     queue_up_non_fatal_config_message()
566     {
567         u_char slotno;
568
569         DPRINTF(("inside queue_up_non_fatal_config_message\n"));
570
571         if (config_error.duplicate_segment) {
572             for (slotno = 0; slotno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++slotno) {
573                 if (config_error.duplicate_segment & (1 << slotno)) {
574                     lm_queue_message(WARNING_MSG, "duplicate segment in lane: to slot: ad",
575                                     'A' + slotno / MAX_SLOT_COUNT,
576                                     slotno % MAX_SLOT_COUNT);
577                 }
578             }
579         }
580
581         if (config_error.missing_segment) {
582             for (slotno = 0; slotno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++slotno) {
583                 if (config_error.missing_segment & (1 << slotno)) {
584                     lm_queue_message(WARNING_MSG, "missing segment for Device Adapter in lane: to slot: ad",
585                                     'A' + slotno / MAX_SLOT_COUNT,
586                                     slotno % MAX_SLOT_COUNT);
587                 }
588             }
589         }
590     }
591 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/lmserver.c

DATE

5/23/89

PAGE #

TIME

6:14:45 pm

6/92

| LINE # | SOURCE TEXT |
|--------|---|
| 601 | } |
| 602 | } |
| 603 | |
| 604 | if (config_error.no_pel_for_dab) { |
| 605 | for (slotno = 0; slotno < MAX_LANE_COUNT; ++slotno) { |
| 606 | if (config_error.no_pel_for_dab & (1 << slotno)) { |
| 607 | lm_queue_message(WARNING_MSG, "Device Adapter in lane: %c slot: %d is not supported by a Pin Electronics Module", |
| 608 | 'A' + slotno / MAX_SLOT_COUNT, |
| 609 | slotno % MAX_SLOT_COUNT); |
| 610 | } |
| 611 | } |
| 612 | |
| 613 | if (config_error.no_pam_for_dab) { |
| 614 | for (slotno = 0; slotno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++slotno) { |
| 615 | if (config_error.no_pam_for_dab & (1 << slotno)) { |
| 616 | lm_queue_message(WARNING_MSG, "Device Adapter in lane: %c slot: %d is not backed by Fast Pattern Memory", |
| 617 | 'A' + slotno / MAX_SLOT_COUNT, |
| 618 | slotno % MAX_SLOT_COUNT); |
| 619 | } |
| 620 | } |
| 621 | |
| 622 | if (config_error.illegal_duplicate_device) { |
| 623 | for (slotno = 0; slotno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++slotno) { |
| 624 | if (config_error.illegal_duplicate_device & (1 << slotno)) { |
| 625 | lm_queue_message(WARNING_MSG, "Non-identical duplicate Device Adapter in lane: %c slot: %d", |
| 626 | 'A' + slotno / MAX_SLOT_COUNT, |
| 627 | slotno % MAX_SLOT_COUNT); |
| 628 | } |
| 629 | } |
| 630 | |
| 631 | if (config_error.device_too_large) { |
| 632 | for (slotno = 0; slotno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++slotno) { |
| 633 | if (config_error.device_too_large & (1 << slotno)) { |
| 634 | lm_queue_message(WARNING_MSG, "Device Adapter in lane: %c slot: %d makes up an illegal device size (> %d units)", |
| 635 | 'A' + slotno / MAX_SLOT_COUNT, |
| 636 | slotno % MAX_SLOT_COUNT, MAX_UNIT_COUNT); |
| 637 | } |
| 638 | } |
| 639 | |
| 640 | if (config_error.illegal_pel_stacking) { |
| 641 | for (slotno = 0; slotno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++slotno) { |
| 642 | if (config_error.illegal_pel_stacking & (1 << slotno)) { |
| 643 | lm_queue_message(WARNING_MSG, "Illegal Pin Electronics Module stacking for Device Adapter in lane: %c slot: %d", |
| 644 | 'A' + slotno / MAX_SLOT_COUNT, |
| 645 | slotno % MAX_SLOT_COUNT, MAX_UNIT_COUNT); |
| 646 | } |
| 647 | } |
| 648 | |
| 649 | } |
| 650 | } |
| 651 | } |
| 652 | } |
| 653 | } |
| 654 | } |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/network.c

DATE 5/23/89
TIME 6:14:46 pm

PAGE #
1/93

| LINE # | SOURCE TEXT |
|--------|--|
| 1 | /* SCCS ID: network.c rev 3.1.1, 4/24/89 at 07:53:22 */ |
| 2 | |
| 3 | #include "common.h" |
| 4 | #include "message.h" |
| 5 | #include "lnetwork.h" |
| 6 | #include "network.h" |
| 7 | |
| 8 | #ifdef DEBUG |
| 9 | #define DPRINTF(x) (void)printf x |
| 10 | #else |
| 11 | #define DPRINTF(x) /* do nothing */ |
| 12 | #endif |
| 13 | |
| 14 | end_queue_message() |
| 15 | { |
| 16 | u_short errors; |
| 17 | u_short warnings; |
| 18 | u_short type; |
| 19 | char str[MAX_MESSAGE]; |
| 20 | char *ptr; |
| 21 | char c; |
| 22 | |
| 23 | /* verify that we have not put anything in the output buffer except |
| 24 | * the answer. |
| 25 | */ |
| 26 | if (LM_BYTES_IN_BUFFER(lm_global_conn_ptr) != 4) { |
| 27 | DPRINTF(("ASSERT: error count is not the first one in out buffer\n")); |
| 28 | } |
| 29 | |
| 30 | lm_message_types(&errors, &warnings); |
| 31 | if (errors+warnings) { |
| 32 | DPRINTF(("there are %d errors and warnings\n", errors+warnings)); |
| 33 | } |
| 34 | |
| 35 | lm_put_int(errors + warnings); |
| 36 | |
| 37 | while (lm_dequeue_message(&type, str) == SUCCESS) { |
| 38 | lm_put_char(type); |
| 39 | |
| 40 | ptr = str; |
| 41 | while (c = *ptr++) { |
| 42 | lm_put_char(c); |
| 43 | } |
| 44 | |
| 45 | DPRINTF(("n")); |
| 46 | lm_put_char('\0'); |
| 47 | } |
| 48 | } |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/nextuser.c

DATE 5/23/89
TIME 6:14:46 pm

PAGE #
1/94

```

1  /* SCCS_ID: nextuser.c rev 3.1, 4/24/89 at 07:53:25 */
2
3  #include "common.h"
4  #include "fifo.h"
5  #include "network.h"
6  #include "cpu.h"
7
8  #ifdef DEBUG
9  #define DPRINTF(x) (void)printf x
10 #else
11 #define DPRINTF(x) /* do nothing */
12 #endif
13
14 extern CONNECTION *table_of_conns[ ];
15 extern u_long lm_dead_user_mask;
16
17 u_short
18 lm_give_me_the_next_user(user)
19 u_char *user;
20 {
21     #ifdef MODELER
22     struct fifo_entry fifo;
23     #endif
24     u_char next_dead_user;
25
26     if (lm_dead_user_mask != 0) {
27         CPU_DISABLE_INTERRUPTS;
28         for (next_dead_user = 0; next_dead_user < MAX_USERS; next_dead_user++) {
29             if (lm_dead_user_mask & (1 << next_dead_user)) {
30                 abort_user_punbar( next_dead_user );
31                 close_connection_for_server( table_of_conns[ next_dead_user ] );
32             }
33         }
34         lm_dead_user_mask = 0;
35         CPU_ENABLE_INTERRUPTS;
36     }
37     #ifdef MODELER
38     u_short ret;
39
40     while (1) {
41         ret = lm_choose_connection(user);
42         if (ret == PENDING) {
43             if (lm_send_reply(table_of_conns[user]) != SUCCESS) {
44                 DPRINTF(("error in lm_send_reply()\n"));
45                 return(FAILURE);
46             }
47         }
48         else {
49             return(ret);
50         }
51     }
52 #else
53     fifo.fifo_no = FI_FIFO;
54     if (fifo_get(fifo) == SUCCESS) {
55         *user = fifo.user;
56         return(SUCCESS);
57     }
58     else {
59         return(FAILURE);
60     }
61 #endif
62 #endif
63 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/pac_util.c

DATE 5/23/89
TIME 6:14:46 pm

PAGE #
1/95

```

1  /* SCCS ID: pac_util.c rev 1.1, 4/24/89 at 07:53.23 */
2
3  #include "common.h"
4  #include "vtx.h"
5  #include "tag.h"
6  #include "tag_def.h"
7  #include "pac.h"
8  #include "pac_def.h"
9
10 extern TAG *tagptr;
11
12 #ifdef DEBUG
13 #define DPRINTF(x) (void)printf x
14 #else
15 #define DPRINTF(x) /* do nothing */
16 #endif
17
18 /*
19  * int backplane_reset()
20  * INPUT: none
21  * OUTPUT: return code = SUCCESS or FAILURE
22  * DESCRIPTION: Resets TMC and all four lanes.
23  */
24 int
25 backplane_reset()
26 {
27     long i;
28
29     /* Initialize Timing Generator */
30     DPRINTF(("inside backplane_reset\n"));
31     *REG(0) = 0x00; /* drive TMC reset and all lane resets */
32     *REG(1) = 0x00;
33     *REG(2) = 0x2c; /* 25 MHz clock */
34     *REG(3) = 0x0f; /* Select divide by 2 */
35     *REG(4) = 0x00; /* Select fast ramps, all lanes */
36     *REG(5) = 0x62; /* Test mode 1, 11ns dlydly */
37     *REG(6) = 0x07; /* 10ns dump, 1st clock */
38     *REG(7) = 0x0f; /* sample width = 1 clock cycle */
39     *REG(16) = 0x07; /* minimum threshold EDGE[0] */
40     *REG(17) = 0x08; /* EDGE[1] */
41     *REG(18) = 0x09; /* EDGE[2] */
42     *REG(19) = 0x0a; /* EDGE[3] */
43     *REG(20) = 0x0b; /* EDGE[4] */
44     *REG(21) = 0x0c; /* EDGE[5] */
45     *REG(22) = 0x07; /* SAMPLE trigger */
46     *REG(23) = 0x07; /* SAMPLE */
47
48     if(pac_clock_off() != SUCCESS)
49         return(FAILURE);
50     if(pac_clock_on() != SUCCESS)
51         return(FAILURE);
52
53     tagptr->tag_reset = 1; /* remove TMC reset */
54
55     if(tag_play(161) != SUCCESS) {
56         DPRINTF(("Power-on play failed.\n"));
57         return(FAILURE);
58     }
59
60     *REG(5) = 0x22; /* No test mode 2, 11ns dlydly */
61
62     if(pac_clock_off() != SUCCESS)
63         return(FAILURE);
64     if(pac_clock_on() != SUCCESS)
65         return(FAILURE);
66
67     DPRINTF(("wait 1\n"));
68     for (i = 0; i < 10000; ++i)
69         ;
70
71     if(pac_clock_off() != SUCCESS)
72         return(FAILURE);
73
74     DPRINTF(("wait 1\n"));
75     for (i = 0; i < 10000; ++i)
76         ;
77
78     *REG(0) = 0x30; /* remove all resets */
79
80     if (tagptr->lane_intr != 0) {
81         DPRINTF(("clearing lane_intr\n"));
82         (void)lm_write_probe(0x8c100280, 0);
83         (void)lm_write_probe(0x9c100280, 0);
84         (void)lm_write_probe(0xac100280, 0);
85         (void)lm_write_probe(0xbc100280, 0);
86     }
87
88     if (tagptr->lane_intr != 0) {
89         DPRINTF(("lane_intr stuck\n"));
90         return(FAILURE);
91     }
92
93     DPRINTF(("backplane reset OK\n"));
94     return(SUCCESS);
95 }
96
97 /*
98  * int pac_clock_on()
99  * INPUT: none
100  * OUTPUT: returns SUCCESS or FAILURE
101  * DESCRIPTION: Turns on pattern clock.
102  */
103 int
104 pac_clock_on()
105 {
106     u_long start;
107
108     DPRINTF(("inside pac_clock_on\n"));
109     tagptr->clock_sync_clear = 1;
110     tagptr->clock_enable = 1;
111     start = lm_time();
112     while(!tagptr->clock_on)
113         ;
114     if((lm_time() - start) > 50) /* 5ms timeout */
115     {
116         DPRINTF(("Unable to turn on clock.\n"));
117         return(FAILURE);
118     }
119 }
120

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/pac_util.c

DATE 5/23/89
TIME 6:14:46 pm

PAGE #
2/96

```
LINE # SOURCE TEXT
121 }
122 return(SUCCESS);
123 }
124
125
126 int pac_clock_off()
127 *
128 * INPUT: none
129 * OUTPUT: returns SUCCESS or FAILURE
130 * DESCRIPTION: Turns off pattern clock.
131 */
132 int
133 pac_clock_off()
134 {
135     u_long start;
136
137     tmgptr->clock_enable = 0;
138     start = lm_time();
139     while(tmgptr->clock_on)
140     {
141         if((lm_time() - start) > 50) /* 5ms timeout */
142         {
143             DPRINTF(("Unable to turn off clock.\n"));
144             return(FAILURE);
145         }
146     }
147     tmgptr->clock_sync_clear1 = 0;
148     return(SUCCESS);
149 }
150
```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/private.c | DATE 5/23/89 | PAGE # 1/97 |
|--|---|------------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCCS ID: private.c rev 3.1, 4/24/89 at 07:53:32 */ | | | |
| 2 | | | | |
| 3 | #include "device.h" | | | |
| 4 | #include "message.h" | | | |
| 5 | #include "hardware.h" | | | |
| 6 | #include "eeprom.h" | | | |
| 7 | #include "laservar.h" | | | |
| 8 | | | | |
| 9 | #define MAX_LINE_LENGTH 256 | | | |
| 10 | | | | |
| 11 | evaluate_private(def_ptr, instance, ident_change, | | | |
| 12 | ident_inconsistent_pins, changed_dac) | | | |
| 13 | DEVICE_SPEC *def_ptr; | | | |
| 14 | INSTANCE_INFO *instance; | | | |
| 15 | PTRN_BITS_LONGWORD *ident_change; | | | |
| 16 | PTRN_BITS_LONGWORD *ident_inconsistent_pins; | | | |
| 17 | u_char *changed_dac; | | | |
| 18 | { | | | |
| 19 | EXTRA_DEVICE_SPEC *extra_def_ptr; | | | |
| 20 | PIN_INFO *pin_info; | | | |
| 21 | PIN_SPEC *pin_def; | | | |
| 22 | UWB_OFFSET *uwb_ptr; | | | |
| 23 | u_long inst_block_number[MAX_LANE_COUNT]; | | | |
| 24 | u_long seg_wnd_addr[MAX_LANE_COUNT]; | | | |
| 25 | u_short eval_index = 0; | | | |
| 26 | i; | | | |
| 27 | short pin_number; | | | |
| 28 | u_char pin_value; | | | |
| 29 | u_char old_pin_value; | | | |
| 30 | u_char junk1; | | | |
| 31 | u_char junk2; | | | |
| 32 | u_char unitno; | | | |
| 33 | u_char wordno; | | | |
| 34 | u_char bitno; | | | |
| 35 | u_char seg_driving_to_s; | | | |
| 36 | { | | | |
| 37 | DPRINTF(("inside evaluate_private\n")); | | | |
| 38 | extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data; | | | |
| 39 | | | | |
| 40 | if (set_edge_and_sample_setting(extra_def_ptr, | | | |
| 41 | (u_long)RESOLUTION_05_NS, | | | |
| 42 | (u_long)MAX_SAMPLE_RANGE) == FAILURE) { | | | |
| 43 | return(FAILURE); | | | |
| 44 | } | | | |
| 45 | | | | |
| 46 | update_ptrn_loaded(instance, def_ptr); | | | |
| 47 | | | | |
| 48 | /* Process the data pins and modify the measurement pattern accordingly */ | | | |
| 49 | pin_number = instance->first_data_pin_index; | | | |
| 50 | instance->first_data_pin_index = -1; | | | |
| 51 | while (pin_number != -1) { | | | |
| 52 | { | | | |
| 53 | uwb_ptr = seg_to_short_offset(pin_number); | | | |
| 54 | unitno = uwb_ptr->unitno; | | | |
| 55 | wordno = uwb_ptr->wordno; | | | |
| 56 | bitno = uwb_ptr->bitno; | | | |
| 57 | pin_info = (instance->pin_info_table[pin_number]); | | | |
| 58 | pin_info->input_pin_is_linked = FALSE; | | | |
| 59 | pin_value = pin_info->old_filtered; | | | |
| 60 | set_pin_value((instance->pin_pin_value, | | | |
| 61 | unitno, wordno, bitno, pin_value); | | | |
| 62 | set_measurement_pattern(instance, def_ptr->pin_table[pin_number], | | | |
| 63 | unitno, wordno, bitno, pin_value, | | | |
| 64 | junk1, junk2); | | | |
| 65 | pin_number = pin_info->next_input_pin_index; | | | |
| 66 | } | | | |
| 67 | | | | |
| 68 | /* Process the eval pins and modify the measurement pattern accordingly */ | | | |
| 69 | pin_number = instance->first_eval_pin_index; | | | |
| 70 | instance->first_eval_pin_index = -1; | | | |
| 71 | while (pin_number != -1) { | | | |
| 72 | { | | | |
| 73 | uwb_ptr = seg_to_short_offset(pin_number); | | | |
| 74 | unitno = uwb_ptr->unitno; | | | |
| 75 | wordno = uwb_ptr->wordno; | | | |
| 76 | bitno = uwb_ptr->bitno; | | | |
| 77 | pin_info = (instance->pin_info_table[pin_number]); | | | |
| 78 | pin_info->input_pin_is_linked = FALSE; | | | |
| 79 | pin_value = pin_info->old_filtered; | | | |
| 80 | set_pin_value((instance->pin_pin_value, | | | |
| 81 | unitno, wordno, bitno, pin_value); | | | |
| 82 | /* Save the EVAL changes to find the output delays */ | | | |
| 83 | if (eval_index < MAX_EVAL_CHANGES) { | | | |
| 84 | gbl_eval_pin_number(eval_index) = pin_number; | | | |
| 85 | gbl_eval_pin_value(eval_index) = pin_value; | | | |
| 86 | ++eval_index; | | | |
| 87 | } | | | |
| 88 | else { | | | |
| 89 | in_queue_message(MEMING_MSG, "internal error: not enough room to store eval changes; delay number might be incorrect"); | | | |
| 90 | } | | | |
| 91 | set_measurement_pattern(instance, def_ptr->pin_table[pin_number], | | | |
| 92 | unitno, wordno, bitno, pin_value, | | | |
| 93 | junk1, junk2); | | | |
| 94 | pin_number = pin_info->next_input_pin_index; | | | |
| 95 | } | | | |
| 96 | | | | |
| 97 | /* If there are any eval changes, then run a measurement cycle */ | | | |
| 98 | if (eval_index != 0) { | | | |
| 99 | { | | | |
| 100 | DPRINTF(("run measurement cycle for eval changes\n")); | | | |
| 101 | | | | |
| 102 | /* Write ENDEND to next to last address */ | | | |
| 103 | write_pattern(instance, | | | |
| 104 | (instance->unit_addr[instance->cur_unit_addr_index + 1 & 1][0], | | | |
| 105 | instance->hndesh_loaded); | | | |
| 106 | | | | |
| 107 | /* Write Consistent out */ | | | |
| 108 | write_pattern(instance, | | | |
| 109 | | | | |
| 110 | | | | |
| 111 | | | | |
| 112 | | | | |
| 113 | | | | |
| 114 | | | | |
| 115 | | | | |
| 116 | | | | |
| 117 | | | | |
| 118 | | | | |
| 119 | | | | |
| 120 | | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/private.c

DATE 5/23/89
TIME 6:14:47 pm

PAGE #
2/98

```

LINE #          SOURCE TEXT
121             *instance->unit_addr[instance->cur_unit_addr_index][0],
122             instance->ptrn_loaded);
123
124             set_seq_end_bit(instance,
125             instance->lane_addr,
126             FALSE,
127             seq_end_addr,
128             inst_block_number);
129
130             if (play_ptrn_seq(instance, (u_long)PTRN_PLAY_TIMEOUT,
131             changed_gsc) == FAILURE) {
132                 return(FAILURE);
133             }
134
135             if (get_result(def_ptr, instance, ident_change,
136             EVAL_EVENT, gbl_eval_pin_number,
137             eval_index, &any_driving_to_z) == FAILURE) {
138                 return(FAILURE);
139             }
140
141             remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
142
143             /* Get the delays from eval pins to outputs and put them in PIN_INFO */
144             for (i = 0; i < eval_index; i++) {
145                 get_delay(def_ptr, instance, ident_change,
146                 ident_inconsistent_pins,
147                 gbl_eval_pin_number[i], gbl_eval_pin_value[i]);
148             }
149
150             clear_ptrn_bits((char *)ident_change,
151             dab_list[instance->dab_info_index]->unit_count);
152
153
154             /* Process the STORE pin changes */
155             for (pin_number = instance->first_store_pin_index,
156             instance->first_store_pin_index = -1,
157             pin_number != -1;
158             pin_number = pin_info->next_input_pin_index) {
159                 update_ptrn_loaded(instance, def_ptr);
160
161                 gbl_eval_pin_number[0] = pin_number;
162
163                 unitno = ipa_to_short_offset(pin_number);
164                 wordno = unitno->wordno;
165                 bitno = unitno->bitno;
166
167                 pin_info = instance->pin_info_table[pin_number];
168                 pin_def = def_ptr->pin_table[pin_number];
169
170                 pin_info->input_pin_is_linked = FALSE;
171                 pin_value = pin_info->old_filtered;
172
173                 if (pin_def->direction == IN) {
174                     /* INPUT STORE change */
175
176                     old_pin_value = read_pin_value(instance->pin_pin_value,
177                     unitno, wordno, bitno);
178
179                     set_pin_value(instance->pin_pin_value,
180                     unitno, wordno, bitno, pin_value);
181
182                     switch (pin_def->clk_format) {
183                         case NRZ:
184                             if (input_pin_transition(old_pin_value, pin_value,
185                             pin_info->uninitialized_pin) == NO_TRANSITION) {
186                                 /* This is an error because the host should have filtered the
187                                 * transition.
188                                 * On second thought it is NOT an error because the host only
189                                 * filters transitions to exactly the same value.
190                                 */
191                                 DPRINTF("No transition on IN STORE NRZ pin %s\n", pin_def->pin_name);
192                                 continue;
193                             }
194
195                             if (pin_value & (LOGIC_0 | LOGIC_50 | LOGIC_Z0))
196                                 reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
197                             else
198                                 set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
199                             break;
200
201                         case R1:
202                             if (input_pin_transition(old_pin_value, pin_value,
203                             pin_info->uninitialized_pin) != RISE_TRANSITION) {
204                                 DPRINTF("No transition on IN STORE R1 pin %s\n", pin_def->pin_name);
205                                 continue;
206                             }
207
208                             /* Enable the R1 clock on the measurement pattern (ptrn_loaded) */
209                             reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
210                             break;
211
212                         case R0:
213                             if (input_pin_transition(old_pin_value, pin_value,
214                             pin_info->uninitialized_pin) != FALL_TRANSITION) {
215                                 DPRINTF("No transition on IN STORE R0 pin %s\n", pin_def->pin_name);
216                                 continue;
217                             }
218
219                             /* Enable the R2 clock on the measurement pattern (ptrn_loaded) */
220                             set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
221                             break;
222
223                         default:
224                             in_queue_message(ERROR_MSG, "internal error: illegal pin type in device.h");
225                             continue;
226                     }
227
228                 } else {
229                     /* IO STORE change */
230                     /* Note: IO store pin can only have DNRZ format */
231
232                     old_pin_value = read_pin_value(instance->last_sample_value,
233                     unitno, wordno, bitno);
234
235

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/private.c

DATE 5/23/89
TIME 6:14:47 pm

PAGE #
3/99

```

LINE # SOURCE TEXT
241 set_pin_value(instance->pin_value,
242 unitno, wordno, bitno, pin_value);
243
244 if (!io_pin_transition(old_pin_value, pin_value,
245 pin_info->uninitialized_pin) == PT_TRANSITION) {
246
247     DPRINTF("No transition on IO STORE pin %s\n", pin_def->pin_name);
248     continue;
249 }
250
251 set_measurement_pattern(instance, pin_def,
252 unitno, wordno, bitno, pin_value,
253 sjunk1, sjunk2);
254
255
256 write_pattern(instance,
257 instance->unit_addr[instance->cur_unit_addr_index + 1][0],
258 instance->inst_block_loaded);
259
260 write_pattern(instance,
261 instance->unit_addr[instance->cur_unit_addr_index][0],
262 instance->ptrn_loaded);
263
264 /* Disable the EL/EL check for the next evaluation */
265 switch (pin_def->ela_format) {
266 case EL:
267     set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
268     break;
269 case R0:
270     reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
271     break;
272 default:
273     break;
274 }
275
276 set_seq_end_bit(instance,
277 instance->lane_addr,
278 FALSE,
279 seq_end_addr,
280 inst_block_number);
281
282 if (play_ptrn_seq(instance, (u_long)PTRN_PLAY_TIMEOUT,
283 changed_dac) == FAILURE) {
284     return(FAILURE);
285 }
286
287 if (get_result(def_ptr, instance, ident_change,
288 STORE_EVENT, gbl_eval_pin_number,
289 1, lazy_driving_to_2) == FAILURE) {
290     return(FAILURE);
291 }
292
293 remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
294
295 get_delay(def_ptr, instance, ident_change, ident_inconsistent_pins,
296 (u_short)pin_number, pin_value);
297
298 clear_ptrn_bits((char *)ident_change,
299 dab_list(instance->dab_info_index->unit_count);
300
301 pin_info->uninitialized_pin = FALSE;
302
303 return(SUCCESS);
304
305
306
307
308 purge_ptrn(instance)
309 INSTANCE_INFO *instance;
310
311 DAB_INFO *dab_ptr;
312 short ptrn_count;
313 u_char dummy_ptrn_count;
314 u_char lane0;
315 u_char i;
316
317 DPRINTF("inside purge_ptrn\n");
318
319 dab_ptr = dab_list(instance->dab_info_index);
320
321 return_all_ptrn_block(instance);
322
323 setup_gbl_dummy_ptrn(dab_ptr);
324
325 for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
326     instance->fb_block_size[lane0] = 0;
327
328     instance->lane_addr[lane0].max_addr = 0;
329     instance->lane_addr[lane0].prev_max_addr = 0;
330     instance->lane_addr[lane0].last_unit_addr = 0;
331     instance->lane_addr[lane0].seq_block_addr = 0;
332 }
333
334 instance->has_history = FALSE;
335 instance->pattern_count = 0;
336 instance->common_pattern_count = 0;
337 instance->static_pattern_count = 0;
338 instance->esab_timing_mode = FALSE;
339
340 if (allocate_initial_block(instance) == FAILURE)
341     return(FAILURE);
342
343 /* set the sequence start address on each lane */
344 for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0)
345     instance->seq_start_addr[lane0] =
346         instance->lane_addr[lane0].max_addr - BLOCK_ADDR_INC;
347
348 /* For PRIVATE devices we only have 2 patterns: ENDING_LOADED and
349 * PTRN_LOADED.
350 */
351 ptrn_count = 2 * dab_ptr->unit_count_per_lane;
352
353 dummy_ptrn_count = 0;
354 if (ptrn_count <= SEQ_END_LATENCY) {
355     dummy_ptrn_count = (SEQ_END_LATENCY - ptrn_count +
356     dab_ptr->unit_count_per_lane) /
357     dab_ptr->unit_count_per_lane;
358 }
359
360

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/private.c

*

DATE 5/23/89
TIME 6:14:47 pm

PAGE #
4/100

```

LINE # SOURCE TEXT
361 for (i = 0; i < dummy_ptrn_count; ++i) {
362     write_pattern(instance,
363         (instance->unit_addr(instance->cur_unit_addr_index)[0],
364         &dummy_ptrn);
365
366     if (grow_pattern(instance) == FAILURE)
367         return(FAILURE);
368 }
369
370 /* allocate 2 patterns for MEMBERS */
371 if (grow_pattern(instance) == FAILURE)
372     return(FAILURE);
373
374 return(SUCCESS);
375 }
376
377 update_ptrn_loaded(instance, def_ptr)
378 INSTANCE_INFO *instance;
379 DEVICE_SPEC *def_ptr;
380 {
381     EXTRA_DEVICE_SPEC *extra_def_ptr;
382     DAB_INFO *dab_ptr;
383     PTRN_BITS_LONGWORD *ptrn_loaded_ptr;
384     PTRN_BITS_LONGWORD *sampled_data_ptr;
385     PTRN_BITS_LONGWORD *sampled_hiz_ptr;
386     PTRN_BITS_LONGWORD *sampled_unk_ptr;
387     PTRN_BITS_LONGWORD *sim_data_ptr;
388     PTRN_BITS_LONGWORD *sim_hiz_ptr;
389     PTRN_BITS_LONGWORD *sim_unk_ptr;
390     PTRN_BITS_LONGWORD *ident_outputs_ptr;
391     PTRN_BITS_LONGWORD *ident_ios_ptr;
392     u_char total_unit;
393     u_char unit;
394     u_char word;
395
396     /* Modify the ptrn_loaded to include the value of the last sample:
397      * for OUTPUT pins --> take the last_sample_value
398      * for IO pins --> take the combination of sim_pin_value and
399      * last_sample_value.
400     */
401
402     extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
403     dab_ptr = dab_list(instance->dab_info_index);
404     total_unit = dab_ptr->unit_count;
405
406     ptrn_loaded_ptr = (PTRN_BITS_LONGWORD *)instance->ptrn_loaded;
407     sampled_data_ptr = (PTRN_BITS_LONGWORD *)
408         instance->last_sample_value.data;
409     sampled_hiz_ptr = (PTRN_BITS_LONGWORD *)
410         instance->last_sample_value.hiz;
411     sampled_unk_ptr = (PTRN_BITS_LONGWORD *)
412         instance->last_sample_value.unknown;
413     sim_data_ptr = (PTRN_BITS_LONGWORD *)
414         instance->sim_pin_value.data;
415     sim_hiz_ptr = (PTRN_BITS_LONGWORD *)
416         instance->sim_pin_value.hiz;
417     sim_unk_ptr = (PTRN_BITS_LONGWORD *)
418         instance->sim_pin_value.unknown;
419     ident_outputs_ptr = extra_def_ptr->ident_outputs;
420     ident_ios_ptr = extra_def_ptr->ident_ios;
421
422     for (unit = 0; unit < total_unit; ++unit) {
423         for (word = 0; word < 3; ++word) {
424
425             /* OUTPUT pins.
426              * Drive the output pins to 0 always.
427              */
428             ptrn_loaded_ptr->word[word] =
429                 (ptrn_loaded_ptr->word[word] & "ident_outputs_ptr->word[word]);
430
431             /* IO pins.
432              * Drive the IO pins to the combination of sim_pin_value and
433              * last_sample_value as follows:
434
435              SIN      ANY      Z      U      ANY
436
437              SAMPLE   ANY      Z      U
438
439              RESULT   $1      $2      $5      $3      $4
440
441              $1 --> use sample data
442              $2 --> use sample data
443              $3 --> use sim data
444              $4 --> use the opposite value of sample data
445              $5 --> use sample data
446
447              ptrn_loaded_ptr->word[word] =
448                  (ptrn_loaded_ptr->word[word] & "ident_ios_ptr->word[word]) |
449                  (ident_ios_ptr->word[word] &
450                  {
451                      /* Result $2 */
452                      ((sampled_hiz_ptr->word[word] & sim_hiz_ptr->word[word]) &
453                      sampled_data_ptr->word[word]) |
454                      /* Result $5 */
455                      ((sampled_hiz_ptr->word[word] & sim_unk_ptr->word[word]) &
456                      sampled_data_ptr->word[word]) |
457                      /* Result $3 */
458                      ((sampled_hiz_ptr->word[word] &
459                      (sim_hiz_ptr->word[word] | sim_unk_ptr->word[word])) &
460                      sim_data_ptr->word[word]) |
461                      /* Result $4 */
462                      (sampled_unk_ptr->word[word] & "sampled_data_ptr->word[word]) |
463                      /* Result $1 */
464                      ((sampled_hiz_ptr->word[word] | sampled_unk_ptr->word[word]) &
465                      sampled_data_ptr->word[word])
466                  });
467
468             ++ptrn_loaded_ptr;
469             ++sampled_data_ptr;
470
471         }
472     }
473 }
474
475 }
476
477 }
478
479 }
480

```

1523

5,353,243

1524

Copyright 1989
Logic Modeling SystemsSOURCE PROGRAM
lm1000/private.cDATE 5/23/89
TIME 6:14:47 pmPAGE #
5/101

| LINE # | SOURCE TEXT |
|--------|----------------------|
| 481 | ++sampled_hiz_ptr; |
| 482 | ++sampled_uak_ptr; |
| 483 | ++sin_data_ptr; |
| 484 | ++sin_hiz_ptr; |
| 485 | ++sin_uak_ptr; |
| 486 | ++ident_outputs_ptr; |
| 487 | ++ident_los_ptr; |
| 488 |) |
| 489 | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/profile.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:47 pm | 1/102 |

```

1  /* SCCS_ID: profile.c rev 1.1, 4/24/89 at 07:53:35 */
2
3  #include "common.h"
4  #include "device.h"
5  #include "message.h"
6  #include "hardware.h"
7  #include "lm_rd_wr.h"
8  #include "mod_err.h"
9  #include "xvar.h"
10 #include "eeprom.h"
11 #include "lserver.h"
12 #include "lnetwork.h"
13 #include "network.h"
14 #include "profile.h"
15
16 PROF_DATA prof_data[MAX_FUNCTION_COUNT];
17 u_short xindex;
18 u_char initialized_profile = FALSE;
19 extern long save_pc;
20
21 profile_clear()
22 {
23     u_short i;
24
25     for (i = 0; i < xindex; ++i) {
26         prof_data[i].count = 0;
27     }
28 }
29
30 profile_incr_count()
31 {
32     register u_long pc;
33     register short start;
34     register short end;
35     register short mid;
36
37     if (initialized_profile == FALSE)
38         return;
39
40     pc = save_pc;
41     start = 0;
42     end = xindex - 1;
43
44     mid = (start + end) / 2;
45     while (start <= end) {
46         if (pc < prof_data[mid].function_addr) {
47             end = mid - 1;
48         }
49         else {
50             start = mid + 1;
51         }
52         mid = (start + end) / 2;
53     }
54
55     while (1) {
56         if ((pc >= prof_data[end].function_addr) &&
57             (pc < prof_data[end+1].function_addr)) {
58             break;
59         }
60         end++;
61     }
62     ++prof_data[end].count;
63 }
64
65 profile_dump_count()
66 {
67     short i;
68     long total;
69     u_long total_count_mark;
70
71     total = 0;
72     total_count_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
73     lm_put_int(0);
74
75     for (i = 0; i < xindex; ++i) {
76         if (prof_data[i].count != 0) {
77             ++total;
78             lm_put_int(prof_data[i].function_addr);
79             lm_put_int(prof_data[i].count);
80         }
81     }
82
83     LM_PUT_LONG_AT_MARK(total_count_mark, lm_global_conn_ptr, total);
84 }
85

```

Copyright 1989

Logic Modeling Systems

HEADER FILE

lm1000/profile.h

DATE

5/23/89

PAGE #

TIME

6:14:47 pm

1/103

| LINE # | HEADER TEXT |
|--------|---|
| 1 | /* SCCS ID: profile.h rev 3.1, 4/24/89 at 07:53:38 */ |
| 2 | |
| 3 | #define MAX_FUNCTION_COUNT 2000 |
| 4 | |
| 5 | typedef struct { |
| 6 | u_long func; _addr; |
| 7 | u_long count; |
| 8 | } PROF_DATA; |
| 9 | |
| 10 | extern PROF_DATA prof_data[MAX_FUNCTION_COUNT]; |
| 11 | extern u_short kindex; |
| 12 | extern u_char initialized_profile; |

Copyright 1989
Logic Modeling SystemsSOURCE PROGRAM
lm1000/profile2.cDATE 5/23/89 PAGE #
TIME 6:14:47 pm 1/104

LINE # SOURCE TEXT

```
1 /* SCCS ID: profile2.c rev 1.1, 4/24/89 at 07:53:41 */
2
3 #include "common.h"
4 #include "profile.h"
5
6 profile_ t()
7 {
8     xindex = 0;
9     prof_data[xindex++].function_addr = 0;
10    prof_data[xindex++].function_addr = 0x00050000;
11    prof_data[xindex++].function_addr = 0x00050026;
12    prof_data[xindex++].function_addr = 0x00050030;
13    prof_data[xindex++].function_addr = 0x00050048;
14    prof_data[xindex++].function_addr = 0x00050076;
15    prof_data[xindex++].function_addr = 0x0005007A;
16    prof_data[xindex++].function_addr = 0x00050092;
17    prof_data[xindex++].function_addr = 0x000500AA;
18    prof_data[xindex++].function_addr = 0x000500DA;
19    prof_data[xindex++].function_addr = 0x000500DE;
20    prof_data[xindex++].function_addr = 0x0005013A;
21    prof_data[xindex++].function_addr = 0x0005014E;
22    prof_data[xindex++].function_addr = 0x0005017A;
23    prof_data[xindex++].function_addr = 0x0005018E;
24    prof_data[xindex++].function_addr = 0x0005019A;
25    prof_data[xindex++].function_addr = 0x000501A8;
26    prof_data[xindex++].function_addr = 0x000501B8;
27    prof_data[xindex++].function_addr = 0x000501E8;
28    prof_data[xindex++].function_addr = 0x00050204;
29    prof_data[xindex++].function_addr = 0x00050230;
30    prof_data[xindex++].function_addr = 0x0005023E;
31    prof_data[xindex++].function_addr = 0x0005024C;
32    prof_data[xindex++].function_addr = 0x00050258;
33    prof_data[xindex++].function_addr = 0x00050268;
34    prof_data[xindex++].function_addr = 0x00050298;
35    prof_data[xindex++].function_addr = 0x000502B4;
36    prof_data[xindex++].function_addr = 0x000502E4;
37    prof_data[xindex++].function_addr = 0x000502FC;
38    prof_data[xindex++].function_addr = 0x00050314;
39    prof_data[xindex++].function_addr = 0x00050330;
40    prof_data[xindex++].function_addr = 0x00050336;
41    prof_data[xindex++].function_addr = 0x00050342;
42    prof_data[xindex++].function_addr = 0x0005038C;
43    prof_data[xindex++].function_addr = 0x00050398;
44    prof_data[xindex++].function_addr = 0x000503A2;
45    prof_data[xindex++].function_addr = 0x000503CC;
46    prof_data[xindex++].function_addr = 0x0005043E;
47    prof_data[xindex++].function_addr = 0x00050480;
48    prof_data[xindex++].function_addr = 0x00050494;
49    prof_data[xindex++].function_addr = 0x000504AA;
50    prof_data[xindex++].function_addr = 0x000504AB;
51    prof_data[xindex++].function_addr = 0x000504B2;
52    prof_data[xindex++].function_addr = 0x000504BA;
53    prof_data[xindex++].function_addr = 0x000504BC;
54    prof_data[xindex++].function_addr = 0x000504F8;
55    prof_data[xindex++].function_addr = 0x00050504;
56    prof_data[xindex++].function_addr = 0x0005073E;
57    prof_data[xindex++].function_addr = 0x0005077F;
58    prof_data[xindex++].function_addr = 0x00050986;
59    prof_data[xindex++].function_addr = 0x00050A02;
60    prof_data[xindex++].function_addr = 0x00050A92;
61    prof_data[xindex++].function_addr = 0x00050AD6;
62    prof_data[xindex++].function_addr = 0x00050B8C;
63    prof_data[xindex++].function_addr = 0x00050C58;
64    prof_data[xindex++].function_addr = 0x00050E56;
65    prof_data[xindex++].function_addr = 0x00050E80;
66    prof_data[xindex++].function_addr = 0x00050FA0;
67    prof_data[xindex++].function_addr = 0x000510DE;
68    prof_data[xindex++].function_addr = 0x000511A2;
69    prof_data[xindex++].function_addr = 0x00051264;
70    prof_data[xindex++].function_addr = 0x0005131E;
71    prof_data[xindex++].function_addr = 0x0005152A;
72    prof_data[xindex++].function_addr = 0x000515EE;
73    prof_data[xindex++].function_addr = 0x000516E8;
74    prof_data[xindex++].function_addr = 0x000516A4;
75    prof_data[xindex++].function_addr = 0x0005192C;
76    prof_data[xindex++].function_addr = 0x00051AA4;
77    prof_data[xindex++].function_addr = 0x00051AD4;
78    prof_data[xindex++].function_addr = 0x00051AE8;
79    prof_data[xindex++].function_addr = 0x00051F34;
80    prof_data[xindex++].function_addr = 0x00051F88;
81    prof_data[xindex++].function_addr = 0x0005201E;
82    prof_data[xindex++].function_addr = 0x00052088;
83    prof_data[xindex++].function_addr = 0x00052080;
84    prof_data[xindex++].function_addr = 0x000520FE;
85    prof_data[xindex++].function_addr = 0x00052144;
86    prof_data[xindex++].function_addr = 0x00052170;
87    prof_data[xindex++].function_addr = 0x000521F4;
88    prof_data[xindex++].function_addr = 0x0005221C;
89    prof_data[xindex++].function_addr = 0x00052244;
90    prof_data[xindex++].function_addr = 0x000522A8;
91    prof_data[xindex++].function_addr = 0x000522E0;
92    prof_data[xindex++].function_addr = 0x00052536;
93    prof_data[xindex++].function_addr = 0x00052786;
94    prof_data[xindex++].function_addr = 0x0005283A;
95    prof_data[xindex++].function_addr = 0x00052A7E;
96    prof_data[xindex++].function_addr = 0x00052C88;
97    prof_data[xindex++].function_addr = 0x00052D4A;
98    prof_data[xindex++].function_addr = 0x00052F68;
99    prof_data[xindex++].function_addr = 0x00052F90;
100   prof_data[xindex++].function_addr = 0x00052FBA;
101   prof_data[xindex++].function_addr = 0x00052FE2;
102   prof_data[xindex++].function_addr = 0x0005301E;
103   prof_data[xindex++].function_addr = 0x0005339C;
104   prof_data[xindex++].function_addr = 0x00053582;
105   prof_data[xindex++].function_addr = 0x000536DA;
106   prof_data[xindex++].function_addr = 0x00053704;
107   prof_data[xindex++].function_addr = 0x0005372C;
108   prof_data[xindex++].function_addr = 0x0005375A;
109   prof_data[xindex++].function_addr = 0x00053C08;
110   prof_data[xindex++].function_addr = 0x00053DEE;
111   prof_data[xindex++].function_addr = 0x00053E0C;
112   prof_data[xindex++].function_addr = 0x00053E1C;
113   prof_data[xindex++].function_addr = 0x00053E38;
114   prof_data[xindex++].function_addr = 0x00053E58;
115   prof_data[xindex++].function_addr = 0x0005418A;
116   prof_data[xindex++].function_addr = 0x000542E4;
117   prof_data[xindex++].function_addr = 0x0005442A;
118   prof_data[xindex++].function_addr = 0x000544EA;
119   prof_data[xindex++].function_addr = 0x0005470E;
120   prof_data[xindex++].function_addr = 0x0005475E;
```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/profile2.c | DATE 5/23/89 | PAGE # 2/105 |
|--|---------------------|-------------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 121 | prof_data(xindex++) | function_addr = 0x000547A8 | | |
| 122 | prof_data(xindex++) | function_addr = 0x000547F6 | | |
| 123 | prof_data(xindex++) | function_addr = 0x0005488A | | |
| 124 | prof_data(xindex++) | function_addr = 0x00054908 | | |
| 125 | prof_data(xindex++) | function_addr = 0x000549F6 | | |
| 126 | prof_data(xindex++) | function_addr = 0x00054A4A | | |
| 127 | prof_data(xindex++) | function_addr = 0x00054B6C | | |
| 128 | prof_data(xindex++) | function_addr = 0x00054C32 | | |
| 129 | prof_data(xindex++) | function_addr = 0x00054D0E | | |
| 130 | prof_data(xindex++) | function_addr = 0x00054E04 | | |
| 131 | prof_data(xindex++) | function_addr = 0x00054F52 | | |
| 132 | prof_data(xindex++) | function_addr = 0x00055024 | | |
| 133 | prof_data(xindex++) | function_addr = 0x00055092 | | |
| 134 | prof_data(xindex++) | function_addr = 0x00055180 | | |
| 135 | prof_data(xindex++) | function_addr = 0x0005527E | | |
| 136 | prof_data(xindex++) | function_addr = 0x0005536A | | |
| 137 | prof_data(xindex++) | function_addr = 0x0005545E | | |
| 138 | prof_data(xindex++) | function_addr = 0x0005554C | | |
| 139 | prof_data(xindex++) | function_addr = 0x0005563A | | |
| 140 | prof_data(xindex++) | function_addr = 0x00055728 | | |
| 141 | prof_data(xindex++) | function_addr = 0x00055816 | | |
| 142 | prof_data(xindex++) | function_addr = 0x00055904 | | |
| 143 | prof_data(xindex++) | function_addr = 0x000559F2 | | |
| 144 | prof_data(xindex++) | function_addr = 0x00055AE2 | | |
| 145 | prof_data(xindex++) | function_addr = 0x00055B50 | | |
| 146 | prof_data(xindex++) | function_addr = 0x00055C3E | | |
| 147 | prof_data(xindex++) | function_addr = 0x00055D2C | | |
| 148 | prof_data(xindex++) | function_addr = 0x00055E1A | | |
| 149 | prof_data(xindex++) | function_addr = 0x00055F08 | | |
| 150 | prof_data(xindex++) | function_addr = 0x00055FF6 | | |
| 151 | prof_data(xindex++) | function_addr = 0x000560E4 | | |
| 152 | prof_data(xindex++) | function_addr = 0x000561D2 | | |
| 153 | prof_data(xindex++) | function_addr = 0x000562C0 | | |
| 154 | prof_data(xindex++) | function_addr = 0x000563AE | | |
| 155 | prof_data(xindex++) | function_addr = 0x0005649C | | |
| 156 | prof_data(xindex++) | function_addr = 0x0005658A | | |
| 157 | prof_data(xindex++) | function_addr = 0x00056678 | | |
| 158 | prof_data(xindex++) | function_addr = 0x00056766 | | |
| 159 | prof_data(xindex++) | function_addr = 0x00056854 | | |
| 160 | prof_data(xindex++) | function_addr = 0x00056942 | | |
| 161 | prof_data(xindex++) | function_addr = 0x00056A30 | | |
| 162 | prof_data(xindex++) | function_addr = 0x00056B1E | | |
| 163 | prof_data(xindex++) | function_addr = 0x00056C0C | | |
| 164 | prof_data(xindex++) | function_addr = 0x00056CFA | | |
| 165 | prof_data(xindex++) | function_addr = 0x00056D88 | | |
| 166 | prof_data(xindex++) | function_addr = 0x00056E76 | | |
| 167 | prof_data(xindex++) | function_addr = 0x00056F64 | | |
| 168 | prof_data(xindex++) | function_addr = 0x00057052 | | |
| 169 | prof_data(xindex++) | function_addr = 0x00057140 | | |
| 170 | prof_data(xindex++) | function_addr = 0x0005722E | | |
| 171 | prof_data(xindex++) | function_addr = 0x0005731C | | |
| 172 | prof_data(xindex++) | function_addr = 0x0005740A | | |
| 173 | prof_data(xindex++) | function_addr = 0x000574F8 | | |
| 174 | prof_data(xindex++) | function_addr = 0x000575E6 | | |
| 175 | prof_data(xindex++) | function_addr = 0x000576D4 | | |
| 176 | prof_data(xindex++) | function_addr = 0x000577C2 | | |
| 177 | prof_data(xindex++) | function_addr = 0x000578B0 | | |
| 178 | prof_data(xindex++) | function_addr = 0x0005799E | | |
| 179 | prof_data(xindex++) | function_addr = 0x00057A8C | | |
| 180 | prof_data(xindex++) | function_addr = 0x00057B7A | | |
| 181 | prof_data(xindex++) | function_addr = 0x00057C68 | | |
| 182 | prof_data(xindex++) | function_addr = 0x00057D56 | | |
| 183 | prof_data(xindex++) | function_addr = 0x00057E44 | | |
| 184 | prof_data(xindex++) | function_addr = 0x00057F32 | | |
| 185 | prof_data(xindex++) | function_addr = 0x00058020 | | |
| 186 | prof_data(xindex++) | function_addr = 0x0005810E | | |
| 187 | prof_data(xindex++) | function_addr = 0x0005819C | | |
| 188 | prof_data(xindex++) | function_addr = 0x0005828A | | |
| 189 | prof_data(xindex++) | function_addr = 0x00058378 | | |
| 190 | prof_data(xindex++) | function_addr = 0x00058466 | | |
| 191 | prof_data(xindex++) | function_addr = 0x00058554 | | |
| 192 | prof_data(xindex++) | function_addr = 0x00058642 | | |
| 193 | prof_data(xindex++) | function_addr = 0x00058730 | | |
| 194 | prof_data(xindex++) | function_addr = 0x0005881E | | |
| 195 | prof_data(xindex++) | function_addr = 0x0005890C | | |
| 196 | prof_data(xindex++) | function_addr = 0x0005899A | | |
| 197 | prof_data(xindex++) | function_addr = 0x00058A88 | | |
| 198 | prof_data(xindex++) | function_addr = 0x00058B76 | | |
| 199 | prof_data(xindex++) | function_addr = 0x00058C64 | | |
| 200 | prof_data(xindex++) | function_addr = 0x00058D52 | | |
| 201 | prof_data(xindex++) | function_addr = 0x00058E40 | | |
| 202 | prof_data(xindex++) | function_addr = 0x00058F2E | | |
| 203 | prof_data(xindex++) | function_addr = 0x0005901C | | |
| 204 | prof_data(xindex++) | function_addr = 0x0005910A | | |
| 205 | prof_data(xindex++) | function_addr = 0x000591F8 | | |
| 206 | prof_data(xindex++) | function_addr = 0x000592E6 | | |
| 207 | prof_data(xindex++) | function_addr = 0x000593D4 | | |
| 208 | prof_data(xindex++) | function_addr = 0x000594C2 | | |
| 209 | prof_data(xindex++) | function_addr = 0x000595B0 | | |
| 210 | prof_data(xindex++) | function_addr = 0x0005969E | | |
| 211 | prof_data(xindex++) | function_addr = 0x0005978C | | |
| 212 | prof_data(xindex++) | function_addr = 0x0005987A | | |
| 213 | prof_data(xindex++) | function_addr = 0x00059968 | | |
| 214 | prof_data(xindex++) | function_addr = 0x00059A56 | | |
| 215 | prof_data(xindex++) | function_addr = 0x00059B44 | | |
| 216 | prof_data(xindex++) | function_addr = 0x00059C32 | | |
| 217 | prof_data(xindex++) | function_addr = 0x00059D20 | | |
| 218 | prof_data(xindex++) | function_addr = 0x00059E0E | | |
| 219 | prof_data(xindex++) | function_addr = 0x00059E9C | | |
| 220 | prof_data(xindex++) | function_addr = 0x00059F8A | | |
| 221 | prof_data(xindex++) | function_addr = 0x0005A078 | | |
| 222 | prof_data(xindex++) | function_addr = 0x0005A166 | | |
| 223 | prof_data(xindex++) | function_addr = 0x0005A254 | | |
| 224 | prof_data(xindex++) | function_addr = 0x0005A342 | | |
| 225 | prof_data(xindex++) | function_addr = 0x0005A430 | | |
| 226 | prof_data(xindex++) | function_addr = 0x0005A51E | | |
| 227 | prof_data(xindex++) | function_addr = 0x0005A60C | | |
| 228 | prof_data(xindex++) | function_addr = 0x0005A69A | | |
| 229 | prof_data(xindex++) | function_addr = 0x0005A788 | | |
| 230 | prof_data(xindex++) | function_addr = 0x0005A876 | | |
| 231 | prof_data(xindex++) | function_addr = 0x0005A964 | | |
| 232 | prof_data(xindex++) | function_addr = 0x0005AA52 | | |
| 233 | prof_data(xindex++) | function_addr = 0x0005AB40 | | |
| 234 | prof_data(xindex++) | function_addr = 0x0005AC2E | | |
| 235 | prof_data(xindex++) | function_addr = 0x0005AD1C | | |
| 236 | prof_data(xindex++) | function_addr = 0x0005AE0A | | |
| 237 | prof_data(xindex++) | function_addr = 0x0005AE98 | | |
| 238 | prof_data(xindex++) | function_addr = 0x0005AF86 | | |
| 239 | prof_data(xindex++) | function_addr = 0x0005A074 | | |
| 240 | prof_data(xindex++) | function_addr = 0x0005A162 | | |

5,353,243

1533

1534

Copyright 1989
Logic Modeling SystemsSOURCE PROGRAM
lm1000/profile2.cDATE 5/23/89
TIME 6:14:47 pm
PAGE # 3/106

LINE # SOURCE TEXT

```

241 prof_data[xindex++].function_addr = 0x00061DD0;
242 prof_data[xindex++].function_addr = 0x00061E56;
243 prof_data[xindex++].function_addr = 0x00061E7C;
244 prof_data[xindex++].function_addr = 0x00062106;
245 prof_data[xindex++].function_addr = 0x00062294;
246 prof_data[xindex++].function_addr = 0x00062342;
247 prof_data[xindex++].function_addr = 0x000623B6;
248 prof_data[xindex++].function_addr = 0x00062852;
249 prof_data[xindex++].function_addr = 0x00062BF4;
250 prof_data[xindex++].function_addr = 0x00062E90;
251 prof_data[xindex++].function_addr = 0x000633FE;
252 prof_data[xindex++].function_addr = 0x00063520;
253 prof_data[xindex++].function_addr = 0x00063558;
254 prof_data[xindex++].function_addr = 0x00063632;
255 prof_data[xindex++].function_addr = 0x000639A0;
256 prof_data[xindex++].function_addr = 0x00063A30;
257 prof_data[xindex++].function_addr = 0x00063AD2;
258 prof_data[xindex++].function_addr = 0x00063BC8;
259 prof_data[xindex++].function_addr = 0x00063C8E;
260 prof_data[xindex++].function_addr = 0x00063CB6;
261 prof_data[xindex++].function_addr = 0x00063D08;
262 prof_data[xindex++].function_addr = 0x00063DB2;
263 prof_data[xindex++].function_addr = 0x000640A0;
264 prof_data[xindex++].function_addr = 0x000641A8;
265 prof_data[xindex++].function_addr = 0x0006439E;
266 prof_data[xindex++].function_addr = 0x000643F8;
267 prof_data[xindex++].function_addr = 0x00064464;
268 prof_data[xindex++].function_addr = 0x0006470A;
269 prof_data[xindex++].function_addr = 0x00065ADA;
270 prof_data[xindex++].function_addr = 0x00065B0C;
271 prof_data[xindex++].function_addr = 0x00065EC6;
272 prof_data[xindex++].function_addr = 0x00066044;
273 prof_data[xindex++].function_addr = 0x000661D4;
274 prof_data[xindex++].function_addr = 0x00066B3A;
275 prof_data[xindex++].function_addr = 0x00066CBA;
276 prof_data[xindex++].function_addr = 0x0006727C;
277 prof_data[xindex++].function_addr = 0x00067444;
278 prof_data[xindex++].function_addr = 0x000674A2;
279 prof_data[xindex++].function_addr = 0x00067500;
280 prof_data[xindex++].function_addr = 0x00067E76;
281 prof_data[xindex++].function_addr = 0x00068040;
282 prof_data[xindex++].function_addr = 0x000681E2;
283 prof_data[xindex++].function_addr = 0x000689DC;
284 prof_data[xindex++].function_addr = 0x000693FC;
285 prof_data[xindex++].function_addr = 0x0006961E;
286 prof_data[xindex++].function_addr = 0x00069D3A;
287 prof_data[xindex++].function_addr = 0x00069E74;
288 prof_data[xindex++].function_addr = 0x0006A000;
289 prof_data[xindex++].function_addr = 0x0006A442;
290 prof_data[xindex++].function_addr = 0x0006A78A;
291 prof_data[xindex++].function_addr = 0x0006A85E;
292 prof_data[xindex++].function_addr = 0x0006A8AA;
293 prof_data[xindex++].function_addr = 0x0006A9CD;
294 prof_data[xindex++].function_addr = 0x0006A9D6;
295 prof_data[xindex++].function_addr = 0x0006AB4C;
296 prof_data[xindex++].function_addr = 0x0006ABE0;
297 prof_data[xindex++].function_addr = 0x0006AC28;
298 prof_data[xindex++].function_addr = 0x0006D854;
299 prof_data[xindex++].function_addr = 0x0006D9DE;
300 prof_data[xindex++].function_addr = 0x0006DD50;
301 prof_data[xindex++].function_addr = 0x0006DF30;
302 prof_data[xindex++].function_addr = 0x0006DFE6;
303 prof_data[xindex++].function_addr = 0x0006E9FC;
304 prof_data[xindex++].function_addr = 0x0006EEFA;
305 prof_data[xindex++].function_addr = 0x0006ED38;
306 prof_data[xindex++].function_addr = 0x0006ED4C;
307 prof_data[xindex++].function_addr = 0x0006EFF8;
308 prof_data[xindex++].function_addr = 0x0006F2FC;
309 prof_data[xindex++].function_addr = 0x0006F35C;
310 prof_data[xindex++].function_addr = 0x00070190;
311 prof_data[xindex++].function_addr = 0x000702E8;
312 prof_data[xindex++].function_addr = 0x00070D9C;
313 prof_data[xindex++].function_addr = 0x00071122;
314 prof_data[xindex++].function_addr = 0x00071210;
315 prof_data[xindex++].function_addr = 0x000712C4;
316 prof_data[xindex++].function_addr = 0x00071690;
317 prof_data[xindex++].function_addr = 0x00071968;
318 prof_data[xindex++].function_addr = 0x00071994;
319 prof_data[xindex++].function_addr = 0x000719E8;
320 prof_data[xindex++].function_addr = 0x00071A08;
321 prof_data[xindex++].function_addr = 0x00071A22;
322 prof_data[xindex++].function_addr = 0x00071B34;
323 prof_data[xindex++].function_addr = 0x00071B6E;
324 prof_data[xindex++].function_addr = 0x00071B8A;
325 prof_data[xindex++].function_addr = 0x00071C16;
326 prof_data[xindex++].function_addr = 0x00071CE2;
327 prof_data[xindex++].function_addr = 0x00071D68;
328 prof_data[xindex++].function_addr = 0x00071D96;
329 prof_data[xindex++].function_addr = 0x00071DD2;
330 prof_data[xindex++].function_addr = 0x00071E44;
331 prof_data[xindex++].function_addr = 0x00071E4A;
332 prof_data[xindex++].function_addr = 0x00071E62;
333 prof_data[xindex++].function_addr = 0x00071F3E;
334 prof_data[xindex++].function_addr = 0x00071F88;
335 prof_data[xindex++].function_addr = 0x000720AA;
336 prof_data[xindex++].function_addr = 0x00072496;
337 prof_data[xindex++].function_addr = 0x00072524;
338 prof_data[xindex++].function_addr = 0x000725CA;
339 prof_data[xindex++].function_addr = 0x00072796;
340 prof_data[xindex++].function_addr = 0x000728FC;
341 prof_data[xindex++].function_addr = 0x0007285E;
342 prof_data[xindex++].function_addr = 0x00073C8C;
343 prof_data[xindex++].function_addr = 0x00073D00;
344 prof_data[xindex++].function_addr = 0x00073DEA;
345 prof_data[xindex++].function_addr = 0x00073E76;
346 prof_data[xindex++].function_addr = 0x00073E94;
347 prof_data[xindex++].function_addr = 0x00073E3E;
348 prof_data[xindex++].function_addr = 0x00073F24;
349 prof_data[xindex++].function_addr = 0x00073FA6;
350 prof_data[xindex++].function_addr = 0x00074014;
351 prof_data[xindex++].function_addr = 0x0007406A;
352 prof_data[xindex++].function_addr = 0x000740FC;
353 prof_data[xindex++].function_addr = 0x0007431C;
354 prof_data[xindex++].function_addr = 0x000743A4;
355 prof_data[xindex++].function_addr = 0x000743F0;
356 prof_data[xindex++].function_addr = 0x00074426;
357 prof_data[xindex++].function_addr = 0x0007444C;
358 prof_data[xindex++].function_addr = 0x00074492;
359 prof_data[xindex++].function_addr = 0x000744C8;
360 prof_data[xindex++].function_addr = 0x000744FE;

```

1536

| | | | | |
|--|--|---|---------------------------------|------------------------|
| Copyright 1989 Logic Modeling Systems | SOURCE PROGRAM lm1000/profile2.c | * | DATE 5/23/89 TIME 6:14:47 pm | PAGE # 4/107 |
|--|--|---|---------------------------------|------------------------|

| LINE # | SOURCE TEXT |
|--------|--|
| 361 | prof_data(xindex++) .function_addr = 0x00074534; |
| 362 | prof_data(xindex++) .function_addr = 0x00074566; |
| 363 | prof_data(xindex++) .function_addr = 0x00074598; |
| 364 | prof_data(xindex++) .function_addr = 0x000745CA; |
| 365 | prof_data(xindex++) .function_addr = 0x000745FC; |
| 366 | prof_data(xindex++) .function_addr = 0x0007462E; |
| 367 | prof_data(xindex++) .function_addr = 0x00074660; |
| 368 | prof_data(xindex++) .function_addr = 0x000747D0; |
| 369 | prof_data(xindex++) .function_addr = 0x00074830; |
| 370 | prof_data(xindex++) .function_addr = 0x0007486E; |
| 371 | prof_data(xindex++) .function_addr = 0x000748B6; |
| 372 | prof_data(xindex++) .function_addr = 0x00074978; |
| 373 | prof_data(xindex++) .function_addr = 0x00074B14; |
| 374 | prof_data(xindex++) .function_addr = 0x00074BFA; |
| 375 | prof_data(xindex++) .function_addr = 0x00074C30; |
| 376 | prof_data(xindex++) .function_addr = 0x00074C96; |
| 377 | prof_data(xindex++) .function_addr = 0x00074CD2; |
| 378 | prof_data(xindex++) .function_addr = 0x00074D0A; |
| 379 | prof_data(xindex++) .function_addr = 0x00074D58; |
| 380 | prof_data(xindex++) .function_addr = 0x00074D90; |
| 381 | prof_data(xindex++) .function_addr = 0x00074DC8; |
| 382 | prof_data(xindex++) .function_addr = 0x00074E04; |
| 383 | prof_data(xindex++) .function_addr = 0x00074E3A; |
| 384 | prof_data(xindex++) .function_addr = 0x00074E74; |
| 385 | prof_data(xindex++) .function_addr = 0x00074F14; |
| 386 | prof_data(xindex++) .function_addr = 0x000750A8; |
| 387 | prof_data(xindex++) .function_addr = 0x000753B2; |
| 388 | prof_data(xindex++) .function_addr = 0x000753C8; |
| 389 | prof_data(xindex++) .function_addr = 0x0007544C; |
| 390 | prof_data(xindex++) .function_addr = 0x000754CC; |
| 391 | prof_data(xindex++) .function_addr = 0x000754FA; |
| 392 | prof_data(xindex++) .function_addr = 0x00075562; |
| 393 | prof_data(xindex++) .function_addr = 0x000755CA; |
| 394 | prof_data(xindex++) .function_addr = 0x000755F4; |
| 395 | prof_data(xindex++) .function_addr = 0x0007567C; |
| 396 | prof_data(xindex++) .function_addr = 0x000756B2; |
| 397 | prof_data(xindex++) .function_addr = 0x00075700; |
| 398 | prof_data(xindex++) .function_addr = 0x00075746; |
| 399 | prof_data(xindex++) .function_addr = 0x00075792; |
| 400 | prof_data(xindex++) .function_addr = 0x00075844; |
| 401 | prof_data(xindex++) .function_addr = 0x00075894; |
| 402 | prof_data(xindex++) .function_addr = 0x00075C6C; |
| 403 | prof_data(xindex++) .function_addr = 0x00075CD4; |
| 404 | prof_data(xindex++) .function_addr = 0x00075D32; |
| 405 | prof_data(xindex++) .function_addr = 0x00075D90; |
| 406 | prof_data(xindex++) .function_addr = 0x00075E1E; |
| 407 | prof_data(xindex++) .function_addr = 0x000764E4; |
| 408 | prof_data(xindex++) .function_addr = 0x00076720; |
| 409 | prof_data(xindex++) .function_addr = 0x00076858; |
| 410 | prof_data(xindex++) .function_addr = 0x00076866; |
| 411 | prof_data(xindex++) .function_addr = 0x000768D4; |
| 412 | prof_data(xindex++) .function_addr = 0x00076B70; |
| 413 | prof_data(xindex++) .function_addr = 0x00076B8E; |
| 414 | prof_data(xindex++) .function_addr = 0x00076B32; |
| 415 | prof_data(xindex++) .function_addr = 0x00076B7E; |
| 416 | prof_data(xindex++) .function_addr = 0x00076BFE; |
| 417 | prof_data(xindex++) .function_addr = 0x0007647E; |
| 418 | prof_data(xindex++) .function_addr = 0x000764CE; |
| 419 | prof_data(xindex++) .function_addr = 0x00076548; |
| 420 | prof_data(xindex++) .function_addr = 0x00076612; |
| 421 | prof_data(xindex++) .function_addr = 0x0007667E; |
| 422 | prof_data(xindex++) .function_addr = 0x00076828; |
| 423 | prof_data(xindex++) .function_addr = 0x0007687A; |
| 424 | prof_data(xindex++) .function_addr = 0x00076A16; |
| 425 | prof_data(xindex++) .function_addr = 0x00076A6C; |
| 426 | prof_data(xindex++) .function_addr = 0x00076A82; |
| 427 | prof_data(xindex++) .function_addr = 0x00076B3C; |
| 428 | prof_data(xindex++) .function_addr = 0x00076B8E; |
| 429 | prof_data(xindex++) .function_addr = 0x00076C3A; |
| 430 | prof_data(xindex++) .function_addr = 0x00076E02; |
| 431 | prof_data(xindex++) .function_addr = 0x0007690E; |
| 432 | prof_data(xindex++) .function_addr = 0x0007698A; |
| 433 | prof_data(xindex++) .function_addr = 0x00076A6C; |
| 434 | prof_data(xindex++) .function_addr = 0x000767D0; |
| 435 | prof_data(xindex++) .function_addr = 0x000767DE; |
| 436 | prof_data(xindex++) .function_addr = 0x000767F4; |
| 437 | prof_data(xindex++) .function_addr = 0x00076A34; |
| 438 | prof_data(xindex++) .function_addr = 0x00076A84; |
| 439 | prof_data(xindex++) .function_addr = 0x00076AEE; |
| 440 | prof_data(xindex++) .function_addr = 0x00076E0A; |
| 441 | prof_data(xindex++) .function_addr = 0x00076AFC; |
| 442 | prof_data(xindex++) .function_addr = 0x000765DC; |
| 443 | prof_data(xindex++) .function_addr = 0x0007663C; |
| 444 | prof_data(xindex++) .function_addr = 0x0007668C; |
| 445 | prof_data(xindex++) .function_addr = 0x000766C6; |
| 446 | prof_data(xindex++) .function_addr = 0x00076AEE; |
| 447 | prof_data(xindex++) .function_addr = 0x000767E0; |
| 448 | prof_data(xindex++) .function_addr = 0x00076830; |
| 449 | prof_data(xindex++) .function_addr = 0x00076856; |
| 450 | prof_data(xindex++) .function_addr = 0x00076890; |
| 451 | prof_data(xindex++) .function_addr = 0x000768B4; |
| 452 | prof_data(xindex++) .function_addr = 0x000768F0; |
| 453 | prof_data(xindex++) .function_addr = 0x00076916; |
| 454 | prof_data(xindex++) .function_addr = 0x0007693C; |
| 455 | prof_data(xindex++) .function_addr = 0x00076962; |
| 456 | prof_data(xindex++) .function_addr = 0x00076990; |
| 457 | prof_data(xindex++) .function_addr = 0x000769C2; |
| 458 | prof_data(xindex++) .function_addr = 0x00076A08; |
| 459 | prof_data(xindex++) .function_addr = 0x00076A62; |

5,353,243

1537

1538

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/profile2.c

DATE

5/23/89

PAGE #

TIME

6:14:47 pm

5/108

LINE # SOURCE TEXT

```
481 prof_data[xindex++].function_addr = 0x0007C4E2;
482 prof_data[xindex++].function_addr = 0x0007C4C6;
483 prof_data[xindex++].function_addr = 0x0007C4E0;
484 prof_data[xindex++].function_addr = 0x0007C53E;
485 prof_data[xindex++].function_addr = 0x0007C5E8;
486 prof_data[xindex++].function_addr = 0x0007C64C;
487 prof_data[xindex++].function_addr = 0x0007C69C;
488 prof_data[xindex++].function_addr = 0x0007C6CA;
489 prof_data[xindex++].function_addr = 0x0007C6E8;
490 prof_data[xindex++].function_addr = 0x0007C702;
491 prof_data[xindex++].function_addr = 0x0007C74C;
492 prof_data[xindex++].function_addr = 0x0007CE2C;
493 prof_data[xindex++].function_addr = 0x0007CE88;
494 prof_data[xindex++].function_addr = 0x0007CEE4;
495 prof_data[xindex++].function_addr = 0x0007D24C;
496 prof_data[xindex++].function_addr = 0x0007E2E6;
497 prof_data[xindex++].function_addr = 0x0007E4E8;
498 prof_data[xindex++].function_addr = 0x0007E5CA;
499 prof_data[xindex++].function_addr = 0x0007E8C6;
500 prof_data[xindex++].function_addr = 0x0007EEAC;
501 prof_data[xindex++].function_addr = 0x0007F1FC;
502 prof_data[xindex++].function_addr = 0x0007F458;
503 prof_data[xindex++].function_addr = 0x0007F756;
504 prof_data[xindex++].function_addr = 0x0007FA54;
505 prof_data[xindex++].function_addr = 0x0007FD52;
506 prof_data[xindex++].function_addr = 0x0007FDDA;
507 prof_data[xindex++].function_addr = 0x0008038C;
508 prof_data[xindex++].function_addr = 0x000804AC;
509 prof_data[xindex++].function_addr = 0x000805FA;
510 prof_data[xindex++].function_addr = 0x000807E0;
511 prof_data[xindex++].function_addr = 0x00081A68;
512 prof_data[xindex++].function_addr = 0x00081BFE;
513 prof_data[xindex++].function_addr = 0x00081DEE;
514 prof_data[xindex++].function_addr = 0x00081E8C;
515 prof_data[xindex++].function_addr = 0x00081ED8;
516 prof_data[xindex++].function_addr = 0x00081FC8;
517 prof_data[xindex++].function_addr = 0x000821E2;
518 prof_data[xindex++].function_addr = 0x00082800;
519 prof_data[xindex++].function_addr = 0x000828B4;
520 prof_data[xindex++].function_addr = 0x000828EA;
521 prof_data[xindex++].function_addr = 0x00082A00;
522 prof_data[xindex++].function_addr = 0x00082AC6;
523 prof_data[xindex++].function_addr = 0x00082C1A;
524 prof_data[xindex++].function_addr = 0x00082F36;
525 prof_data[xindex++].function_addr = 0x00083252;
526 prof_data[xindex++].function_addr = 0x00083620;
527 prof_data[xindex++].function_addr = 0x000837CC;
528 prof_data[xindex++].function_addr = 0x00083CF8;
529 prof_data[xindex++].function_addr = 0x00083E4A;
530 prof_data[xindex++].function_addr = 0x00083F74;
531 prof_data[xindex++].function_addr = 0x00083B1C;
532 prof_data[xindex++].function_addr = 0x00083B78;
533 prof_data[xindex++].function_addr = 0x00083CCE;
534 prof_data[xindex++].function_addr = 0x00083D48;
535 prof_data[xindex++].function_addr = 0x00083E1A;
536 prof_data[xindex++].function_addr = 0x00083E2A;
537 prof_data[xindex++].function_addr = 0x00083E4E;
538 prof_data[xindex++].function_addr = 0x00083EAD;
539 prof_data[xindex++].function_addr = 0x00083EAD;
540 prof_data[xindex++].function_addr = 0x00083E9A;
541 prof_data[xindex++].function_addr = 0x00083E74;
542 prof_data[xindex++].function_addr = 0x00083E97;
543 prof_data[xindex++].function_addr = 0x00083E42;
544 prof_data[xindex++].function_addr = 0x00083E28;
545 prof_data[xindex++].function_addr = 0x00083704;
546 prof_data[xindex++].function_addr = 0x000837CC;
547 prof_data[xindex++].function_addr = 0x0008372C;
548 prof_data[xindex++].function_addr = 0x00083776;
549 prof_data[xindex++].function_addr = 0x00083746;
550 prof_data[xindex++].function_addr = 0x00083706;
551 prof_data[xindex++].function_addr = 0x0008378E;
552 prof_data[xindex++].function_addr = 0x00083764;
553 prof_data[xindex++].function_addr = 0x0008370A;
554 prof_data[xindex++].function_addr = 0x00083788;
555 prof_data[xindex++].function_addr = 0x000837D8;
556 prof_data[xindex++].function_addr = 0x00083712;
557 prof_data[xindex++].function_addr = 0x00083774;
558 prof_data[xindex++].function_addr = 0x00083704;
559 prof_data[xindex++].function_addr = 0x0008373C;
560 prof_data[xindex++].function_addr = 0x00083764;
561 prof_data[xindex++].function_addr = 0x00083774;
562 prof_data[xindex++].function_addr = 0x00083784;
563 prof_data[xindex++].function_addr = 0x00083794;
564 prof_data[xindex++].function_addr = 0x000837A4;
565 prof_data[xindex++].function_addr = 0x000837B4;
566 prof_data[xindex++].function_addr = 0x000837C4;
567 prof_data[xindex++].function_addr = 0x000837D4;
568 prof_data[xindex++].function_addr = 0x000837E4;
569 prof_data[xindex++].function_addr = 0x000837F4;
570 prof_data[xindex++].function_addr = 0x00083804;
571 prof_data[xindex++].function_addr = 0x00083814;
572 prof_data[xindex++].function_addr = 0x00083824;
573 prof_data[xindex++].function_addr = 0x00083834;
574 prof_data[xindex++].function_addr = 0x00083844;
575 prof_data[xindex++].function_addr = 0x00083854;
576 prof_data[xindex++].function_addr = 0x00083864;
577 prof_data[xindex++].function_addr = 0x00083874;
578 prof_data[xindex++].function_addr = 0x00083882;
579 prof_data[xindex++].function_addr = 0x00083892;
580 prof_data[xindex++].function_addr = 0x000838A2;
581 prof_data[xindex++].function_addr = 0x000838C6;
582 prof_data[xindex++].function_addr = 0x00083824;
583 prof_data[xindex++].function_addr = 0x000838B6;
584 prof_data[xindex++].function_addr = 0x0008384C;
585 prof_data[xindex++].function_addr = 0x00083820;
586 prof_data[xindex++].function_addr = 0x00083830;
587 prof_data[xindex++].function_addr = 0x0008382E;
588 prof_data[xindex++].function_addr = 0x0008388E;
589 prof_data[xindex++].function_addr = 0x0008390E;
590 prof_data[xindex++].function_addr = 0x000839CA;
591 prof_data[xindex++].function_addr = 0x00083BC4;
592 prof_data[xindex++].function_addr = 0x00083C1A;
593 prof_data[xindex++].function_addr = 0x00083E0D;
594 prof_data[xindex++].function_addr = 0x00083F68;
595 prof_data[xindex++].function_addr = 0x00083906;
596 prof_data[xindex++].function_addr = 0x00083984;
597 prof_data[xindex++].function_addr = 0x000839C0;
598 prof_data[xindex++].function_addr = 0x0008392E;
599 prof_data[xindex++].function_addr = 0x0008398E;
600 prof_data[xindex++].function_addr = 0x0008392A;
```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/profile2.c | DATE 5/23/89 | PAGE # 6/109 |
|--|---------------------|-------------------------------------|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 601 | prof_data(xindex++) | .function_addr = 0x0008927E, | | |
| 602 | prof_data(xindex++) | .function_addr = 0x0008931C, | | |
| 603 | prof_data(xindex++) | .function_addr = 0x000893B8, | | |
| 604 | prof_data(xindex++) | .function_addr = 0x00089354, | | |
| 605 | prof_data(xindex++) | .function_addr = 0x000893EC, | | |
| 606 | prof_data(xindex++) | .function_addr = 0x00089600, | | |
| 607 | prof_data(xindex++) | .function_addr = 0x00089612, | | |
| 608 | prof_data(xindex++) | .function_addr = 0x000896CE, | | |
| 609 | prof_data(xindex++) | .function_addr = 0x0008A19C, | | |
| 610 | prof_data(xindex++) | .function_addr = 0x0008A312, | | |
| 611 | prof_data(xindex++) | .function_addr = 0x0008A354, | | |
| 612 | prof_data(xindex++) | .function_addr = 0x0008A376, | | |
| 613 | prof_data(xindex++) | .function_addr = 0x0008A3D2, | | |
| 614 | prof_data(xindex++) | .function_addr = 0x0008AE7C, | | |
| 615 | prof_data(xindex++) | .function_addr = 0x0008B292, | | |
| 616 | prof_data(xindex++) | .function_addr = 0x0008B384, | | |
| 617 | prof_data(xindex++) | .function_addr = 0x0008B37C, | | |
| 618 | prof_data(xindex++) | .function_addr = 0x0008B456, | | |
| 619 | prof_data(xindex++) | .function_addr = 0x0008B7CE, | | |
| 620 | prof_data(xindex++) | .function_addr = 0x0008B87A, | | |
| 621 | prof_data(xindex++) | .function_addr = 0x0008B916, | | |
| 622 | prof_data(xindex++) | .function_addr = 0x0008BA3A, | | |
| 623 | prof_data(xindex++) | .function_addr = 0x0008BAC2, | | |
| 624 | prof_data(xindex++) | .function_addr = 0x0008BC6A, | | |
| 625 | prof_data(xindex++) | .function_addr = 0x0008BD92, | | |
| 626 | prof_data(xindex++) | .function_addr = 0x0008BE3E, | | |
| 627 | prof_data(xindex++) | .function_addr = 0x0008BF72, | | |
| 628 | prof_data(xindex++) | .function_addr = 0x0008C1B4, | | |
| 629 | prof_data(xindex++) | .function_addr = 0x0008C39E, | | |
| 630 | prof_data(xindex++) | .function_addr = 0x0008C410, | | |
| 631 | prof_data(xindex++) | .function_addr = 0x0008C4EE, | | |
| 632 | prof_data(xindex++) | .function_addr = 0x0008C6D0, | | |
| 633 | prof_data(xindex++) | .function_addr = 0x0008C7B6, | | |
| 634 | prof_data(xindex++) | .function_addr = 0x0008D19C, | | |
| 635 | prof_data(xindex++) | .function_addr = 0x0008D7AA, | | |
| 636 | prof_data(xindex++) | .function_addr = 0x0008DE76, | | |
| 637 | prof_data(xindex++) | .function_addr = 0x0008E1D6, | | |
| 638 | prof_data(xindex++) | .function_addr = 0x0008E746, | | |
| 639 | prof_data(xindex++) | .function_addr = 0x0008E78A, | | |
| 640 | prof_data(xindex++) | .function_addr = 0x0008E7A2, | | |
| 641 | prof_data(xindex++) | .function_addr = 0x0008E7E6, | | |
| 642 | prof_data(xindex++) | .function_addr = 0x0008E816, | | |
| 643 | prof_data(xindex++) | .function_addr = 0x0008EC38, | | |
| 644 | prof_data(xindex++) | .function_addr = 0x0008EDC2, | | |
| 645 | prof_data(xindex++) | .function_addr = 0x0008EE14, | | |
| 646 | prof_data(xindex++) | .function_addr = 0x0008EE40, | | |
| 647 | prof_data(xindex++) | .function_addr = 0x0008EEAA, | | |
| 648 | prof_data(xindex++) | .function_addr = 0x0008EF20, | | |
| 649 | prof_data(xindex++) | .function_addr = 0x0008EFAC, | | |
| 650 | prof_data(xindex++) | .function_addr = 0x0008F08C, | | |
| 651 | prof_data(xindex++) | .function_addr = 0x0008F150, | | |
| 652 | prof_data(xindex++) | .function_addr = 0x0008F27A, | | |
| 653 | prof_data(xindex++) | .function_addr = 0x0008F376, | | |
| 654 | prof_data(xindex++) | .function_addr = 0x0008F396, | | |
| 655 | prof_data(xindex++) | .function_addr = 0x0008F3BE, | | |
| 656 | prof_data(xindex++) | .function_addr = 0x0008F3A4, | | |
| 657 | prof_data(xindex++) | .function_addr = 0x0008F4D6, | | |
| 658 | prof_data(xindex++) | .function_addr = 0x0008F7E0, | | |
| 659 | prof_data(xindex++) | .function_addr = 0x0008FA82, | | |
| 660 | prof_data(xindex++) | .function_addr = 0x0008FAB8, | | |
| 661 | prof_data(xindex++) | .function_addr = 0x0008FB36, | | |
| 662 | prof_data(xindex++) | .function_addr = 0x0008F7C4, | | |
| 663 | prof_data(xindex++) | .function_addr = 0x00090308, | | |
| 664 | prof_data(xindex++) | .function_addr = 0x00090324, | | |
| 665 | prof_data(xindex++) | .function_addr = 0x00090340, | | |
| 666 | prof_data(xindex++) | .function_addr = 0x000903E0, | | |
| 667 | prof_data(xindex++) | .function_addr = 0x000904C4, | | |
| 668 | prof_data(xindex++) | .function_addr = 0x0009054E, | | |
| 669 | prof_data(xindex++) | .function_addr = 0x00090652, | | |
| 670 | prof_data(xindex++) | .function_addr = 0x00090770, | | |
| 671 | prof_data(xindex++) | .function_addr = 0x0009085E, | | |
| 672 | prof_data(xindex++) | .function_addr = 0x000909F2, | | |
| 673 | prof_data(xindex++) | .function_addr = 0x00090AAE, | | |
| 674 | prof_data(xindex++) | .function_addr = 0x00090ABE, | | |
| 675 | prof_data(xindex++) | .function_addr = 0x00090A7E, | | |
| 676 | prof_data(xindex++) | .function_addr = 0x00090868, | | |
| 677 | prof_data(xindex++) | .function_addr = 0x00090C10, | | |
| 678 | prof_data(xindex++) | .function_addr = 0x00090E76, | | |
| 679 | prof_data(xindex++) | .function_addr = 0x00091008, | | |
| 680 | prof_data(xindex++) | .function_addr = 0x00091380, | | |
| 681 | prof_data(xindex++) | .function_addr = 0x000913DE, | | |
| 682 | prof_data(xindex++) | .function_addr = 0x000914C8, | | |
| 683 | prof_data(xindex++) | .function_addr = 0x00091566, | | |
| 684 | prof_data(xindex++) | .function_addr = 0x00091510, | | |
| 685 | prof_data(xindex++) | .function_addr = 0x0009151A, | | |
| 686 | prof_data(xindex++) | .function_addr = 0x00091540, | | |
| 687 | prof_data(xindex++) | .function_addr = 0x00091492, | | |
| 688 | prof_data(xindex++) | .function_addr = 0x00091488, | | |
| 689 | prof_data(xindex++) | .function_addr = 0x00091760, | | |
| 690 | prof_data(xindex++) | .function_addr = 0x000917DC, | | |
| 691 | prof_data(xindex++) | .function_addr = 0x00091ADE, | | |
| 692 | prof_data(xindex++) | .function_addr = 0x00091B24, | | |
| 693 | prof_data(xindex++) | .function_addr = 0x00091B4A, | | |
| 694 | prof_data(xindex++) | .function_addr = 0x00091B5A, | | |
| 695 | prof_data(xindex++) | .function_addr = 0x00091B48, | | |
| 696 | prof_data(xindex++) | .function_addr = 0x00091B8C, | | |
| 697 | prof_data(xindex++) | .function_addr = 0x00091C1C, | | |
| 698 | prof_data(xindex++) | .function_addr = 0x00091C3C, | | |
| 699 | prof_data(xindex++) | .function_addr = 0x00091C6C, | | |
| 700 | prof_data(xindex++) | .function_addr = 0x00091E28, | | |
| 701 | prof_data(xindex++) | .function_addr = 0x00091F88, | | |
| 702 | prof_data(xindex++) | .function_addr = 0x00092086, | | |
| 703 | prof_data(xindex++) | .function_addr = 0x000921C0, | | |
| 704 | prof_data(xindex++) | .function_addr = 0x000922DC, | | |
| 705 | prof_data(xindex++) | .function_addr = 0x0009231C, | | |
| 706 | prof_data(xindex++) | .function_addr = 0x00092348, | | |
| 707 | prof_data(xindex++) | .function_addr = 0x0009237E, | | |
| 708 | prof_data(xindex++) | .function_addr = 0x0009239E, | | |
| 709 | prof_data(xindex++) | .function_addr = 0x00092370, | | |
| 710 | prof_data(xindex++) | .function_addr = 0x00092434, | | |
| 711 | prof_data(xindex++) | .function_addr = 0x000924E2, | | |
| 712 | prof_data(xindex++) | .function_addr = 0x00092558, | | |
| 713 | prof_data(xindex++) | .function_addr = 0x00092582, | | |
| 714 | prof_data(xindex++) | .function_addr = 0x000925F8, | | |
| 715 | prof_data(xindex++) | .function_addr = 0x0009269C, | | |
| 716 | prof_data(xindex++) | .function_addr = 0x000926D2, | | |
| 717 | prof_data(xindex++) | .function_addr = 0x0009274E, | | |
| 718 | prof_data(xindex++) | .function_addr = 0x00092804, | | |
| 719 | prof_data(xindex++) | .function_addr = 0x00092862, | | |
| 720 | prof_data(xindex++) | .function_addr = 0x000928DE, | | |

5,353,243

1541

1542

Copyright 1989
Logic Modeling SystemsSOURCE PROGRAM
lm1000/profile2.cDATE 5/23/89 PAGE #
TIME 6:14:47 pm 7/110

| LINE # | SOURCE TEXT |
|--------|---|
| 721 | prof_data[xindex++].function_addr = 0x0009295A, |
| 722 | prof_data[xindex++].function_addr = 0x00092B1A, |
| 723 | prof_data[xindex++].function_addr = 0x00092B7A, |
| 724 | prof_data[xindex++].function_addr = 0x00092BD4, |
| 725 | prof_data[xindex++].function_addr = 0x00092C2A, |
| 726 | prof_data[xindex++].function_addr = 0x00092F66, |
| 727 | prof_data[xindex++].function_addr = 0x00093122, |
| 728 | prof_data[xindex++].function_addr = 0x000931C4, |
| 729 | prof_data[xindex++].function_addr = 0x00093250, |
| 730 | prof_data[xindex++].function_addr = 0x00093286, |
| 731 | prof_data[xindex++].function_addr = 0x000934D8, |
| 732 | prof_data[xindex++].function_addr = 0x0009356C, |
| 733 | prof_data[xindex++].function_addr = 0x00093570, |
| 734 | prof_data[xindex++].function_addr = 0x00093B60, |
| 735 | prof_data[xindex++].function_addr = 0x00094020, |
| 736 | prof_data[xindex++].function_addr = 0x00094062, |
| 737 | prof_data[xindex++].function_addr = 0x00094114, |
| 738 | prof_data[xindex++].function_addr = 0x0009414C, |
| 739 | prof_data[xindex++].function_addr = 0x00094AF4, |
| 740 | prof_data[xindex++].function_addr = 0x00094B10, |
| 741 | prof_data[xindex++].function_addr = 0x00094B2C, |
| 742 | prof_data[xindex++].function_addr = 0x00094B60, |
| 743 | prof_data[xindex++].function_addr = 0x00094BA0, |
| 744 | prof_data[xindex++].function_addr = 0x00094D88, |
| 745 | prof_data[xindex++].function_addr = 0x00094E7C, |
| 746 | prof_data[xindex++].function_addr = 0x00094F3E, |
| 747 | prof_data[xindex++].function_addr = 0x00094FBA, |
| 748 | prof_data[xindex++].function_addr = 0x00095020, |
| 749 | prof_data[xindex++].function_addr = 0x00095056, |
| 750 | prof_data[xindex++].function_addr = 0x0009507A, |
| 751 | prof_data[xindex++].function_addr = 0x000950BA, |
| 752 | prof_data[xindex++].function_addr = 0x000950DE, |
| 753 | prof_data[xindex++].function_addr = 0x00095210, |
| 754 | prof_data[xindex++].function_addr = 0x00095260, |
| 755 | prof_data[xindex++].function_addr = 0x000954CA, |
| 756 | prof_data[xindex++].function_addr = 0x000954E2, |
| 757 | prof_data[xindex++].function_addr = 0x000954FA, |
| 758 | prof_data[xindex++].function_addr = 0x0009551C, |
| 759 | prof_data[xindex++].function_addr = 0x0009553A, |
| 760 | prof_data[xindex++].function_addr = 0x00095554, |
| 761 | prof_data[xindex++].function_addr = 0x0009558E, |
| 762 | prof_data[xindex++].function_addr = 0x0009564E, |
| 763 | prof_data[xindex++].function_addr = 0x000956E2, |
| 764 | prof_data[xindex++].function_addr = 0x00095740, |
| 765 | prof_data[xindex++].function_addr = 0x000957BA, |
| 766 | prof_data[xindex++].function_addr = 0x00095824, |
| 767 | prof_data[xindex++].function_addr = 0x00095866, |
| 768 | prof_data[xindex++].function_addr = 0x000958E8, |
| 769 | prof_data[xindex++].function_addr = 0x00095980, |
| 770 | prof_data[xindex++].function_addr = 0x00095A18, |
| 771 | prof_data[xindex++].function_addr = 0x00095B34, |
| 772 | prof_data[xindex++].function_addr = 0x00095B9E, |
| 773 | prof_data[xindex++].function_addr = 0x00095C46, |
| 774 | prof_data[xindex++].function_addr = 0x00095CDE, |
| 775 | prof_data[xindex++].function_addr = 0x00095CF4, |
| 776 | prof_data[xindex++].function_addr = 0x00095D5C, |
| 777 | prof_data[xindex++].function_addr = 0x00095D72, |
| 778 | prof_data[xindex++].function_addr = 0x00095E22, |
| 779 | prof_data[xindex++].function_addr = 0x00095E6C, |
| 780 | prof_data[xindex++].function_addr = 0x00095E88, |
| 781 | prof_data[xindex++].function_addr = 0x00095EB8, |
| 782 | prof_data[xindex++].function_addr = 0x00095ED8, |
| 783 | prof_data[xindex++].function_addr = 0x00095F10, |
| 784 | prof_data[xindex++].function_addr = 0x00095F34, |
| 785 | prof_data[xindex++].function_addr = 0x00095F84, |
| 786 | prof_data[xindex++].function_addr = 0x00095FCC, |
| 787 | prof_data[xindex++].function_addr = 0x00096094, |
| 788 | prof_data[xindex++].function_addr = 0x000960C8, |
| 789 | prof_data[xindex++].function_addr = 0x00096114, |
| 790 | prof_data[xindex++].function_addr = 0x00096144, |
| 791 | prof_data[xindex++].function_addr = 0x00096174, |
| 792 | prof_data[xindex++].function_addr = 0x000961CC, |
| 793 | prof_data[xindex++].function_addr = 0x00096374, |
| 794 | prof_data[xindex++].function_addr = 0x0009639C, |
| 795 | prof_data[xindex++].function_addr = 0x0009651C, |
| 796 | prof_data[xindex++].function_addr = 0x000965D0, |
| 797 | prof_data[xindex++].function_addr = 0x000965F8, |
| 798 | prof_data[xindex++].function_addr = 0x000965B0, |
| 799 | prof_data[xindex++].function_addr = 0x000965D4, |
| 800 | prof_data[xindex++].function_addr = 0x00096808, |
| 801 | prof_data[xindex++].function_addr = 0x00096C0C, |
| 802 | prof_data[xindex++].function_addr = 0x00096CB4, |
| 803 | prof_data[xindex++].function_addr = 0x00096D9E, |
| 804 | prof_data[xindex++].function_addr = 0x00096E74, |
| 805 | prof_data[xindex++].function_addr = 0x00097080, |
| 806 | prof_data[xindex++].function_addr = 0x000970E8, |
| 807 | prof_data[xindex++].function_addr = 0x0009713C, |
| 808 | prof_data[xindex++].function_addr = 0x0009726C, |
| 809 | prof_data[xindex++].function_addr = 0x0009748C, |
| 810 | prof_data[xindex++].function_addr = 0x000974F8, |
| 811 | prof_data[xindex++].function_addr = 0x00097504, |
| 812 | prof_data[xindex++].function_addr = 0x00097640, |
| 813 | prof_data[xindex++].function_addr = 0x000976AE, |
| 814 | prof_data[xindex++].function_addr = 0x0009765C, |
| 815 | prof_data[xindex++].function_addr = 0x00097678, |
| 816 | prof_data[xindex++].function_addr = 0x000976A0, |
| 817 | prof_data[xindex++].function_addr = 0x000976AE, |
| 818 | prof_data[xindex++].function_addr = 0x000976CC, |
| 819 | prof_data[xindex++].function_addr = 0x000976E2, |
| 820 | prof_data[xindex++].function_addr = 0x000976FA, |
| 821 | prof_data[xindex++].function_addr = 0x00097712, |
| 822 | prof_data[xindex++].function_addr = 0x0009772C, |
| 823 | prof_data[xindex++].function_addr = 0x00097744, |
| 824 | prof_data[xindex++].function_addr = 0x00097768, |
| 825 | prof_data[xindex++].function_addr = 0x0009778A, |
| 826 | prof_data[xindex++].function_addr = 0x000977B8, |
| 827 | prof_data[xindex++].function_addr = 0x000977E4, |
| 828 | prof_data[xindex++].function_addr = 0x0009780C, |
| 829 | prof_data[xindex++].function_addr = 0x00097832, |
| 830 | prof_data[xindex++].function_addr = 0x00097850, |
| 831 | prof_data[xindex++].function_addr = 0x00097876, |
| 832 | prof_data[xindex++].function_addr = 0x0009789C, |
| 833 | prof_data[xindex++].function_addr = 0x000978CC, |
| 834 | prof_data[xindex++].function_addr = 0x000978F0, |
| 835 | prof_data[xindex++].function_addr = 0x00097914, |
| 836 | prof_data[xindex++].function_addr = 0x00097938, |
| 837 | prof_data[xindex++].function_addr = 0x0009795E, |
| 838 | prof_data[xindex++].function_addr = 0x0009798E, |
| 839 | prof_data[xindex++].function_addr = 0x0009799E, |
| 840 | prof_data[xindex++].function_addr = 0x000979B0, |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/profile2.c | DATE 5/23/89 | PAGE # 8/111 |
|--|---|-------------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 841 | prof_data[xindex++].function_addr = 0x000979DE; | | | |
| 842 | prof_data[xindex++].function_addr = 0x00097A5E; | | | |
| 843 | prof_data[xindex++].function_addr = 0x00097B5C; | | | |
| 844 | prof_data[xindex++].function_addr = 0x00097B86; | | | |
| 845 | prof_data[xindex++].function_addr = 0x00097BD0; | | | |
| 846 | prof_data[xindex++].function_addr = 0x00097C24; | | | |
| 847 | prof_data[xindex++].function_addr = 0x00097C2A; | | | |
| 848 | prof_data[xindex++].function_addr = 0x00097D5A; | | | |
| 849 | prof_data[xindex++].function_addr = 0x00097DA8; | | | |
| 850 | prof_data[xindex++].function_addr = 0x00097E8C; | | | |
| 851 | prof_data[xindex++].function_addr = 0x00097FE6; | | | |
| 852 | prof_data[xindex++].function_addr = 0x00097FF0; | | | |
| 853 | prof_data[xindex++].function_addr = 0x00098118; | | | |
| 854 | prof_data[xindex++].function_addr = 0x00098184; | | | |
| 855 | prof_data[xindex++].function_addr = 0x00098184; | | | |
| 856 | prof_data[xindex++].function_addr = 0x00098214; | | | |
| 857 | prof_data[xindex++].function_addr = 0x00098242; | | | |
| 858 | prof_data[xindex++].function_addr = 0x0009838C; | | | |
| 859 | prof_data[xindex++].function_addr = 0x000983C0; | | | |
| 860 | prof_data[xindex++].function_addr = 0x00098464; | | | |
| 861 | prof_data[xindex++].function_addr = 0x0009855A; | | | |
| 862 | prof_data[xindex++].function_addr = 0x000985AC; | | | |
| 863 | prof_data[xindex++].function_addr = 0x000985F8; | | | |
| 864 | prof_data[xindex++].function_addr = 0x00098642; | | | |
| 865 | prof_data[xindex++].function_addr = 0x00098646; | | | |
| 866 | prof_data[xindex++].function_addr = 0x000986F6; | | | |
| 867 | prof_data[xindex++].function_addr = 0x00098724; | | | |
| 868 | prof_data[xindex++].function_addr = 0x0009874E; | | | |
| 869 | prof_data[xindex++].function_addr = 0x00098790; | | | |
| 870 | prof_data[xindex++].function_addr = 0x000987D0; | | | |
| 871 | prof_data[xindex++].function_addr = 0x000987D8; | | | |
| 872 | prof_data[xindex++].function_addr = 0x00098816; | | | |
| 873 | prof_data[xindex++].function_addr = 0x00098828; | | | |
| 874 | prof_data[xindex++].function_addr = 0x0009883A; | | | |
| 875 | prof_data[xindex++].function_addr = 0x0009884A; | | | |
| 876 | prof_data[xindex++].function_addr = 0x0009885C; | | | |
| 877 | prof_data[xindex++].function_addr = 0x000988F0; | | | |
| 878 | prof_data[xindex++].function_addr = 0x00098936; | | | |
| 879 | prof_data[xindex++].function_addr = 0x0009898C; | | | |
| 880 | prof_data[xindex++].function_addr = 0x000989F8; | | | |
| 881 | prof_data[xindex++].function_addr = 0x00098A78; | | | |
| 882 | prof_data[xindex++].function_addr = 0x00098B0C; | | | |
| 883 | prof_data[xindex++].function_addr = 0x00098B5C; | | | |
| 884 | prof_data[xindex++].function_addr = 0x00098BA0; | | | |
| 885 | prof_data[xindex++].function_addr = 0x00098BFE; | | | |
| 886 | prof_data[xindex++].function_addr = 0x00098C54; | | | |
| 887 | prof_data[xindex++].function_addr = 0x00098CD0; | | | |
| 888 | prof_data[xindex++].function_addr = 0x00098D60; | | | |
| 889 | prof_data[xindex++].function_addr = 0x00098D7E; | | | |
| 890 | prof_data[xindex++].function_addr = 0x00099398; | | | |
| 891 | prof_data[xindex++].function_addr = 0x000993A4; | | | |
| 892 | prof_data[xindex++].function_addr = 0xffffffff; | | | |
| 893 | initialized_profile = TRUE; | | | |
| 894 | | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/ptrnhist.c

DATE 5/23/89
TIME 6:14:48 pm

PAGE #
1/112

```

LINE # SOURCE TEXT
1  /* SCCS ID: ptrnhist.c rev 3.1, 4/24/89 at 07:53:47 */
2
3  #include "device.h"
4  #include "message.h"
5  #include "history.h"
6  #include "a_i.h"
7  #include "lserver.h"
8  #include "protass.h"
9  #include "lnetwork.h"
10 #include "network.h"
11
12 #define MAX_LINE_LENGTH 128
13
14 #ifdef DEBUG
15 extern u_char loop_till_key;
16 extern u_long debug_key;
17 extern u_long debug_char;
18 #endif
19
20 evaluate_1_bit_per_pin(def_ptr, instance,
21                      ident_inconsistent_pins,
22                      temp_steady_state_result,
23                      ident_change,
24                      replay_count,
25                      changed_dec)
26
27 DEVICE_SPEC *def_ptr;
28 INSTANCE_INFO *instance;
29 PTRN_BITS LONGWORD *ident_inconsistent_pins;
30 FULL_VALUE *temp_steady_state_result;
31 PTRN_BITS LONGWORD *ident_change;
32 u_char replay_count;
33 u_char changed_dec;
34
35 EXTRA_DEVICE_SPEC *extra_def_ptr;
36 PIN_INFO *pin_info;
37 PIN_SPEC *pin_def;
38 PTRN_BITS consistent_set(MAX_UNIT_COUNT);
39 UWB_OFFSET *uwb_ptr;
40 u_long inst_block_number(MAX_LANE_COUNT);
41 u_long seq_wnd_addr(MAX_LANE_COUNT);
42 timer_t timer;
43 u_short eval_index = 0;
44 u_short i;
45 u_short pin_number;
46 u_char pin_value;
47 u_char old_pin_value;
48 u_char lcyddb_modified = FALSE;
49 u_char lcyddb_modified = FALSE;
50 u_char unitno;
51 u_char wordno;
52 u_char bitno;
53 u_char total_unit;
54 u_char any_driving_to_1;
55
56 DPRINTF(("inside evaluate_1_bit_per_pin\n"));
57
58 /* The timeout value is set with the assumption that we are running at
59 * 15MHz ("bus period").
60 * timeout = pattern play time + 1 second for feedback
61 * = pattern_count * 8 microsec + 125000 * 8 microsec
62 * = (pattern_count + 125000) * 8 microsec
63 * = (pattern_count + 125000) * 8 / 1000 millisecc
64 */
65 timeout = (instance->pattern_count + PTRN_COUNT_FUDGE_FACTOR) >> 7;
66
67 extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
68
69 if (set_edge_and_sample_setting(extra_def_ptr,
70                                (u_long)RESOLUTION_05_NS,
71                                (u_long)MAX_SAMPLE_RANGE) == FAILURE)
72     return(FAILURE);
73
74 total_unit = dab_list(instance->dab_info_index)->unit_count;
75
76 if (instance->is_fault == TRUE) {
77     /* If it is a FAULT then always write the LCYDBS and LCYDBS */
78     write_pattern(instance,
79                  &instance->lcyddb_addr[0],
80                  instance->lcyddb_loaded);
81
82     write_pattern(instance,
83                  &instance->lcyddb_addr[0],
84                  instance->lcyddb_loaded);
85 }
86 else {
87     /* If it is an INSTANCE then write the LCYDBS and LCYDBS if this
88     * instance has any faults.
89     */
90     if (instance->fault_count != 0) {
91         write_pattern(instance,
92                      &instance->lcyddb_addr[0],
93                      instance->lcyddb_loaded);
94
95         write_pattern(instance,
96                      &instance->lcyddb_addr[0],
97                      instance->lcyddb_loaded);
98     }
99 }
100
101 /* Process the data pins and modify the measurement pattern accordingly */
102 pin_number = instance->first_data_pin_index;
103 instance->first_data_pin_index = -1;
104 while (pin_number != -1) {
105     uwb_ptr = apn_to_short_offset(pin_number);
106     unitno = uwb_ptr->unitno;
107     wordno = uwb_ptr->wordno;
108     bitno = uwb_ptr->bitno;
109
110     pin_info = &instance->pin_info_table[pin_number];
111     pin_info->input_pin_is_linked = FALSE;
112     pin_value = pin_info->old_filtered;
113
114     set_pin_value(&instance->pin_value,
115                  unitno, wordno, bitno, pin_value);
116
117     set_measurement_pattern(instance, &def_ptr->pin_table[pin_number],
118                             unitno, wordno, bitno, pin_value,
119                             lcyddb_modified, lcyddb_modified);
120 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/ptrnhist.c

DATE 5/23/89
TIME 6:14:48 pm

PAGE #
2/113

```

121  SOURCE TEXT
122
123  pin_number = pin_info->next_input_pin_index;
124  }
125
126  /* Process the eval pins and modify the measurement pattern accordingly */
127  pin_number = instance->first_eval_pin_index;
128  instance->first_eval_pin_index = -1;
129  while (pin_number != -1) {
130
131      uwb_ptr = ipn_to_short_offset(pin_number);
132      unitno = uwb_ptr->unitno;
133      wordno = uwb_ptr->wordno;
134      bitno = uwb_ptr->bitno;
135
136      pin_info = instance->pin_info_table[pin_number];
137      pin_info->input_pin_is_linked = FALSE;
138      pin_value = pin_info->old_filtered;
139
140      set_pin_value(instance->pin_value,
141                  unitno, wordno, bitno, pin_value);
142
143      /* Save the EVAL changes to find the output delays */
144      if (eval_index < MAX_EVAL_CHANGES) {
145          gbl_eval_pin_number[eval_index] = pin_number;
146          gbl_eval_pin_value[eval_index] = pin_value;
147          ++eval_index;
148      }
149      else {
150          lm_queue_message(WARNING_MSG, "internal error: not enough room to store eval changes; delay number might be incorrect");
151      }
152
153      set_measurement_pattern(instance, &def_ptr->pin_table[pin_number],
154                          unitno, wordno, bitno, pin_value,
155                          &lcychdb_modified, &lcycmdb_modified);
156
157      pin_number = pin_info->next_input_pin_index;
158  }
159
160  /* If there are any eval changes, then run a measurement cycle */
161  if (eval_index != 0) {
162
163      DPRINTF(("run measurement cycle for eval changes\n"));
164
165      if (lcychdb_modified == TRUE) {
166          write_pattern(instance,
167                      instance->lcychdb_addr[0],
168                      instance->lcychdb_loaded);
169          lcychdb_modified = FALSE;
170      }
171
172      if (lcycmdb_modified == TRUE) {
173          write_pattern(instance,
174                      instance->lcycmdb_addr[0],
175                      instance->lcycmdb_loaded);
176          lcycmdb_modified = FALSE;
177      }
178
179      write_pattern(instance,
180                  instance->unit_addr[instance->cur_unit_addr_index][0],
181                  instance->ptrs_loaded);
182
183      set_seq_end_bit(instance,
184                  instance->last_addr,
185                  FALSE,
186                  seq_end_addr,
187                  inst_block_number);
188
189      if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
190          remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
191          return(FAILURE);
192      }
193
194      if (get_result(def_ptr, instance, idest_change,
195                  EVAL_EVENT, gbl_eval_pin_number,
196                  eval_index, &any_driving_to_z) == FAILURE) {
197          remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
198          return(FAILURE);
199      }
200
201      for (i = 0; i < replay_count; ++i) {
202          if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
203              remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
204              return(FAILURE);
205          }
206
207          read_magic_full_sample_reg(instance, temp_steady_state_result);
208          check_consistency(instance,
209                          instance->last_sample_value,
210                          temp_steady_state_result,
211                          idest_inconsistent_pins,
212                          idest_change,
213                          total_unit);
214      }
215
216      #ifdef DEBUG
217      if (loop_till_key == TRUE) {
218          printf("looping in eval instance. EVAL changes\n");
219          debug_key = 0;
220          while (debug_key == 0) {
221              if (play_ptrn_seq(instance,
222                          timeout, &changed_dac) == FAILURE) {
223                  remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
224                  return(FAILURE);
225              }
226
227              if (debug_char == (u_long)'t') {
228                  loop_till_key = FALSE;
229                  printf("stop loop\n");
230              }
231              debug_key = 0;
232          }
233      #endif
234
235      if (any_driving_to_z == TRUE) {
236          /* Use 2 bits per pin -> hard drive those IO STORE pins which
237             change from driving to Z.
238          */
239
240          if (def_ptr->device_type == PUBLIC) {
241              /* Write ENDEND_LOADED to previous pattern address.

```

```

LINE # SOURCE TEXT
241  * For PRIVATE devices we don't have to do anything since
242  * RMODE will always be written when we do the next evaluation.
243  */
244  write_ptrns(instance,
245             (instance->unit_addr[
246              instance->cur_unit_addr_index + 1 & 1][0],
247              instance->hmdemb_loaded);
248
249  write_ptrns(instance,
250             (instance->unit_addr[instance->cur_unit_addr_index][0],
251              instance->last_consistent_set);
252
253  if (grow_ptrns(instance) == FAILURE)
254      return(FAILURE);
255
256  }
257
258  remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
259
260
261  /* Get the delays from eval pins to outputs and put them in PIN_INFO */
262  for (i = 0; i < eval_index; ++i) {
263      get_delay(def_ptr, instance, ideat_change,
264              ideat_inconsistent_pins,
265              gbl_eval_pin_number[i], gbl_eval_pin_value[i]);
266  }
267
268  add_inconsistent_pins(instance, ideat_inconsistent_pins);
269
270  clear_ptrns_bits((char *)ideat_change,
271                  dab_list(instance->dab_info_index->unit_count);
272
273  clear_ptrns_bits((char *)ideat_inconsistent_pins,
274                  dab_list(instance->dab_info_index->unit_count);
275
276  }
277
278  /* Process the STORE pin changes */
279  for (pin_number = instance->first_store_pin_index,
280       instance->first_store_pin_index = -1,
281       pin_number != -1,
282       pin_number = pin_info->next_input_pin_index) {
283      gbl_eval_pin_number[0] = pin_number;
284
285      uwb_ptr = uwb_ptr - abs_offset(pin_number);
286      unitno = uwb_ptr->unitno;
287      wordno = uwb_ptr->wordno;
288      bitno = uwb_ptr->bitno;
289
290      pin_info = instance->pin_info_table[pin_number];
291      pin_def = def_ptr->pin_table[pin_number];
292
293      pin_info->input_pin_is_linked = FALSE;
294      pin_value = pin_info->old_filtered;
295
296      if (pin_def->direction == IN) {
297          /* INPUT STORE change */
298
299          old_pin_value = read_pin_value(instance->pin_value,
300                                       unitno, wordno, bitno);
301
302          set_pin_value(instance->pin_value,
303                      unitno, wordno, bitno, pin_value);
304
305          switch (pin_def->clk_format) {
306              case DNRZ:
307
308                  if (input_pin_transition(old_pin_value, pin_value,
309                                           pin_info->uninitialized_pin) == NO_TRANSITION) {
310                      /* This is an error because the host should have filtered the
311                       * transition.
312                       * On success thought it is NOT an error because the host only
313                       * filters transitions to exactly the same value.
314                       */
315                      DPRINTF(("No transition on IN STORE DNRZ pin %s", pin_def->pin_name));
316
317                      continue;
318                  }
319
320                  if (pin_value & (LOGIC_0 | LOGIC_S0 | LOGIC_Z0))
321                      reset_ptrns_bit(instance->ptrns_loaded[unitno], wordno, bitno);
322                  else
323                      set_ptrns_bit(instance->ptrns_loaded[unitno], wordno, bitno);
324
325                  break;
326
327              case R1:
328                  if (input_pin_transition(old_pin_value, pin_value,
329                                           pin_info->uninitialized_pin) != RISE_TRANSITION) {
330                      DPRINTF(("No transition on IN STORE R1 pin %s", pin_def->pin_name));
331                      continue;
332                  }
333
334                  /* Disable the R1 clock on the measurement pattern (ptrns_loaded) */
335                  set_ptrns_bit(instance->ptrns_loaded[unitno], wordno, bitno);
336
337                  break;
338
339              case R0:
340                  if (input_pin_transition(old_pin_value, pin_value,
341                                           pin_info->uninitialized_pin) != FALL_TRANSITION) {
342                      DPRINTF(("No transition on IN STORE R0 pin %s", pin_def->pin_name));
343                      continue;
344                  }
345
346                  /* Disable the RZ clock on the measurement pattern (ptrns_loaded) */
347                  reset_ptrns_bit(instance->ptrns_loaded[unitno], wordno, bitno);
348
349                  break;
350
351              default:
352                  lm_queue_message(ERROR_MSG, "internal error: illegal pin type in device.b");
353                  continue;
354          }
355      }
356  }
357
358  /* IO STORE change */
359  /* Note: IO store pin can only have DNRZ format */
360  old_pin_value = read_pin_value(instance->last_sample_value,

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/ptrnhist.c

DATE 5/23/89
TIME 6:14:48 pm

PAGE #
4/115

```

LINE # SOURCE TEXT
361         unitno, wordno, bitno);
362
363     set_pin_value(instance->pin_value,
364                 unitno, wordno, bitno, pin_value);
365
366     if (io_pin_transition(old_pin_value, pin_value,
367                         pin_info->uninitialized_pin) == NO_TRANSITION) {
368
369         DPRINTF(("No transition on IO STORE pin %s", pin_def->pin_name));
370
371         continue;
372     }
373
374     set_measurement_pattern(instance, pin_def,
375                             unitno, wordno, bitno, pin_value,
376                             &lcycdb_modified, &lcycmdb_modified);
377
378
379     if (lcycdb_modified == TRUE) {
380         write_pattern(instance,
381                     instance->lcycdb_addr[0],
382                     instance->lcycdb_loaded);
383         lcycdb_modified = FALSE;
384     }
385
386     if (lcycmdb_modified == TRUE) {
387         write_pattern(instance,
388                     instance->lcycmdb_addr[0],
389                     instance->lcycmdb_loaded);
390         lcycmdb_modified = FALSE;
391     }
392
393     calculate_consistent_set(instance, def_ptr, consistent_set);
394
395     switch (pin_def->clk_format) {
396     case R1:
397         reset_ptrn_bit(&consistent_set[unitno], wordno, bitno);
398         break;
399     case R0:
400         set_ptrn_bit (&consistent_set[unitno], wordno, bitno);
401         break;
402     default:
403         break;
404     }
405
406     /* save the consistent set */
407     copy_ptrn_bits((char *)consistent_set,
408                  (char *)instance->last_consistent_set,
409                  dab_list(instance->dab_info_index->unit_count);
410
411     if (instance->use_2_bit_per_pin == TRUE) {
412         switch_to_2_bit_per_pin(instance);
413     }
414
415     write_pattern(instance,
416                 instance->unit_addr(instance->cur_unit_addr_index)[0],
417                 consistent_set);
418
419     if (grow_pattern(instance) == FAILURE)
420         return(FAILURE);
421
422     write_pattern(instance,
423                 instance->unit_addr(instance->cur_unit_addr_index)[0],
424                 instance->ptrn_loaded);
425
426     set_seq_end_bit(instance,
427                     instance->last_addr,
428                     FALSE,
429                     seq_end_addr,
430                     inst_block_number);
431
432     if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
433         remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
434         return(FAILURE);
435     }
436
437     if (get_result(def_ptr, instance, ident_change,
438                 STORE_EVENT, gbl_eval_pin_number,
439                 1, &any_driving_to_z) == FAILURE) {
440         remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
441         return(FAILURE);
442     }
443
444     for (i = 0; i < replay_count; ++i) {
445         if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
446             remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
447             return(FAILURE);
448         }
449
450         read_magic_full_sample_reg(instance, temp_steady_state_result);
451         check_consistency(instance,
452                             instance->last_sample_value,
453                             temp_steady_state_result,
454                             ident_inconsistent_pins,
455                             ident_change,
456                             total_unit);
457     }
458
459     #ifdef DEBUG
460     if (loop_till_key == TRUE) {
461         printf("looping in eval instance. STORE change...\n");
462         debug_key = 0;
463         while (debug_key == 0) {
464             if (play_ptrn_seq(instance,
465                             timeout, &changed_dac) == FAILURE) {
466                 remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
467                 return(FAILURE);
468             }
469
470             if (debug_char == (u_long)'t') {
471                 loop_till_key = FALSE;
472                 printf("stop loop\n");
473             }
474             debug_key = 0;
475         }
476     }
477     #endif
478
479     if (any_driving_to_z == TRUE) {

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/ptrnhist.c

DATE 5/23/89 PAGE #
TIME 6:14:48 pm 5/116

```

LINE # SOURCE TEXT
481 /* Use 2 bits per pin -> hard drive those IO STOR pins which
482 change from driving to 2.
483 */
484
485 if (def_ptr->device_type == PUBLIC) {
486     /* Write some LOADED to previous pattern addr...
487     * For PRIVATE devices we don't have to do anything since
488     * ROMS will always be written when we do the next evaluation.
489     */
490     write_pattern(instance,
491         (instance->unit_addr[
492             instance->cur_unit_addr_index + 1 & 1][0],
493             instance->hndemb_loaded);
494
495     write_pattern(instance,
496         (instance->unit_addr[instance->cur_unit_addr_index][0],
497             instance->last_consistent_set);
498
499     if (grow_pattern(instance) == FAILURE)
500         return(FAILURE);
501 }
502
503 remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
504
505 get_delay(def_ptr, instance, ident_change,
506     ident_inconsistent_pins,
507     (u_short)pin_number, pin_value);
508
509 add_inconsistent_pins(instance, ident_inconsistent_pins);
510
511 clear_ptrn_bits((char *)ident_change,
512     dab_list(instance->dab_info_index)->unit_count);
513
514 clear_ptrn_bits((char *)ident_inconsistent_pins,
515     dab_list(instance->dab_info_index)->unit_count);
516
517 pin_info->uninitialized_pin = FALSE;
518 }
519
520 return(SUCCESS);
521 }
522
523 set_measurement_pattern(instance, pin_def, unitno, wordno, bitno, pin_value,
524     lcyddb_modified, lcyddb_modified);
525
526 INSTANCE INFO
527 PIN_SPEC
528 u_char
529 u_char
530 u_char
531 u_char
532 u_char
533 u_char
534 {
535     u_char measured_value;
536
537     if (pin_value & (LOGIC_0 | LOGIC_S0))
538         reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
539     else if (pin_value & (LOGIC_1 | LOGIC_S1))
540         set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
541     else {
542         measured_value = read_pin_value(instance->last_sample_value,
543             unitno, wordno, bitno);
544         if (measured_value & (LOGIC_0 | LOGIC_S0))
545             reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
546         else if (measured_value & (LOGIC_1 | LOGIC_S1))
547             set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
548         else {
549             /* Simulator value is I/O and measured value is U-Drive
550              * this pin to the opposite value.
551              */
552             toggle_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
553         }
554     }
555
556     if (instance->definition->device_type == PRIVATE) {
557         if (pin_value & (LOGIC_0 | LOGIC_S0 | LOGIC_S1)) {
558             reset_ptrn_bit(instance->hndemb_loaded[unitno], wordno, bitno);
559         }
560         else {
561             set_ptrn_bit(instance->hndemb_loaded[unitno], wordno, bitno);
562         }
563         return;
564     }
565
566     if (pin_def->direction == IO) {
567         if (pin_value & (LOGIC_0 | LOGIC_S0 | LOGIC_S1)) {
568             switch (pin_def->last_cyc_drive) {
569                 case R_DRIVE:
570                     reset_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);
571                     *lcyddb_modified = TRUE;
572                     break;
573                 case M_DRIVE:
574                     reset_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);
575                     *lcyddb_modified = TRUE;
576                     break;
577                 case RM_DRIVE:
578                     reset_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);
579                     *lcyddb_modified = TRUE;
580                     *lcyddb_modified = TRUE;
581                     break;
582                 case NO_DRIVE:
583                     default:
584                         break;
585             }
586         }
587         else {
588             switch (pin_def->last_cyc_drive) {
589                 case R_DRIVE:
590                     set_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);
591                     *lcyddb_modified = TRUE;
592                     break;
593                 case M_DRIVE:
594                     set_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);
595                     *lcyddb_modified = TRUE;
596                     break;
597                 case RM_DRIVE:
598                     set_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);
599                     set_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);
600

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/ptrnhist.c

DATE 07/23/89
TIME 6:14:48 pm

PAGE #
6/117

LINE # SOURCE TEXT

```

601  *leycmds_modified = TRUE;
602  *leycmds_modified = TRUE;
603  break;
604  case NO_DRIVE:
605  default:
606  break;
607  }
608  }
609  }
610  }
611  }
612  calculate_consistent_set(instance, def_ptr, consistent_set_ptr)
613  {
614  INSTANCE_INFO *instance;
615  DEVICE_SPEC *def_ptr;
616  PTRN_BITS_LONGWORD *consistent_set_ptr;
617  {
618  EXTRA_DEVICE_SPEC *extra_def_ptr;
619  DAB_INFO *dab_ptr;
620  PTRN_BITS_LONGWORD *ptrn_loaded_ptr;
621  PTRN_BITS_LONGWORD *sampled_data_ptr;
622  PTRN_BITS_LONGWORD *sampled_hiz_ptr;
623  PTRN_BITS_LONGWORD *sampled_unk_ptr;
624  PTRN_BITS_LONGWORD *sim_data_ptr;
625  PTRN_BITS_LONGWORD *sim_hiz_ptr;
626  PTRN_BITS_LONGWORD *sim_unk_ptr;
627  PTRN_BITS_LONGWORD *ident_inputs_ptr;
628  PTRN_BITS_LONGWORD *ident_outputs_ptr;
629  PTRN_BITS_LONGWORD *ident_los_ptr;
630  u_char total_unit;
631  u_char unit;
632  u_char word;
633  }
634  /* Calculate consistent set:
635  * for INPUT pins --> take the sim_pin_value (or ptrn_loaded)
636  * disable R1/R2 clocks
637  * for OUTPUT pins --> take the last_sample_value
638  * for IO pins --> take the combination of sim_pin_value and
639  * last_sample_value.
640  */
641  extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
642  dab_ptr = dab_list(instance->dab_info_index);
643  total_unit = dab_ptr->unit_count;
644  ptrn_loaded_ptr = (PTRN_BITS_LONGWORD *)instance->ptrn_loaded;
645  sampled_data_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.data;
646  sampled_hiz_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.hiz;
647  sampled_unk_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.unknown;
648  sim_data_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.data;
649  sim_hiz_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.hiz;
650  sim_unk_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.unknown;
651  ident_inputs_ptr = extra_def_ptr->ident_inputs;
652  ident_outputs_ptr = extra_def_ptr->ident_outputs;
653  ident_los_ptr = extra_def_ptr->ident_los;
654  for (unit = 0; unit < total_unit; ++unit) {
655  for (word = 0; word < J; ++word) {
656  /* INPUT pins.
657  * The ptrn_loaded already contains the clock and the data bits
658  * set correctly before this step, so just copy it. We cannot
659  * use the sim_pin_value because the clock (R1/R2) might not be
660  * set correctly. For example if a pin is an R1 pin and we have
661  * seen the falling transition but not the rising transition yet.
662  * At that point the sim_pin_value will be set to 0. If some
663  * other state pin change value and we use sim_pin_value to
664  * build the consistent set, then the consistent set will have
665  * that clock enabled (which is wrong !!!).
666  */
667  consistent_set_ptr->word[word] =
668  ptrn_loaded_ptr->word[word] & ident_inputs_ptr->word[word];
669  /* OUTPUT pins.
670  * Drive the output pins to the last sampled value.
671  */
672  consistent_set_ptr->word[word] |=
673  sampled_data_ptr->word[word] & ident_outputs_ptr->word[word];
674  /* IO pins.
675  * Drive the IO pins to the combination of sim_pin_value and
676  * last_sample_value as follows:
677  *
678  *
679  *
680  *
681  *
682  *
683  *
684  *
685  *
686  *
687  *
688  *
689  *
690  *
691  *
692  *
693  *
694  *
695  *
696  *
697  *
698  *
699  *
700  *
701  *
702  *
703  *
704  *
705  *
706  *
707  *
708  *
709  *
710  *
711  *
712  *
713  *
714  *
715  *
716  *
717  *
718  *
719  *
720  *
721  *
722  *

```

| | | | | |
|--------|-----|-----|-----|-----|
| SIM | ANY | Z | U | ANY |
| SAMPLE | ANY | Z | U | |
| RESULT | \$1 | \$2 | \$5 | \$3 |

```

721  *
722  *
723  *
724  *
725  *
726  *
727  *
728  *
729  *
730  *
731  *
732  *
733  *
734  *
735  *
736  *
737  *
738  *
739  *
740  *
741  *
742  *
743  *
744  *
745  *
746  *
747  *
748  *
749  *
750  *
751  *
752  *
753  *
754  *
755  *
756  *
757  *
758  *
759  *
760  *
761  *
762  *
763  *
764  *
765  *
766  *
767  *
768  *
769  *
770  *
771  *
772  *
773  *
774  *
775  *
776  *
777  *
778  *
779  *
780  *
781  *
782  *
783  *
784  *
785  *
786  *
787  *
788  *
789  *
790  *
791  *
792  *
793  *
794  *
795  *
796  *
797  *
798  *
799  *
800  *
801  *
802  *
803  *
804  *
805  *
806  *
807  *
808  *
809  *
810  *
811  *
812  *
813  *
814  *
815  *
816  *
817  *
818  *
819  *
820  *
821  *
822  *
823  *
824  *
825  *
826  *
827  *
828  *
829  *
830  *
831  *
832  *
833  *
834  *
835  *
836  *
837  *
838  *
839  *
840  *
841  *
842  *
843  *
844  *
845  *
846  *
847  *
848  *
849  *
850  *
851  *
852  *
853  *
854  *
855  *
856  *
857  *
858  *
859  *
860  *
861  *
862  *
863  *
864  *
865  *
866  *
867  *
868  *
869  *
870  *
871  *
872  *
873  *
874  *
875  *
876  *
877  *
878  *
879  *
880  *
881  *
882  *
883  *
884  *
885  *
886  *
887  *
888  *
889  *
890  *
891  *
892  *
893  *
894  *
895  *
896  *
897  *
898  *
899  *
900  *
901  *
902  *
903  *
904  *
905  *
906  *
907  *
908  *
909  *
910  *
911  *
912  *
913  *
914  *
915  *
916  *
917  *
918  *
919  *
920  *
921  *
922  *
923  *
924  *
925  *
926  *
927  *
928  *
929  *
930  *
931  *
932  *
933  *
934  *
935  *
936  *
937  *
938  *
939  *
940  *
941  *
942  *
943  *
944  *
945  *
946  *
947  *
948  *
949  *
950  *
951  *
952  *
953  *
954  *
955  *
956  *
957  *
958  *
959  *
960  *
961  *
962  *
963  *
964  *
965  *
966  *
967  *
968  *
969  *
970  *
971  *
972  *
973  *
974  *
975  *
976  *
977  *
978  *
979  *
980  *
981  *
982  *
983  *
984  *
985  *
986  *
987  *
988  *
989  *
990  *
991  *
992  *
993  *
994  *
995  *
996  *
997  *
998  *
999  *
1000  *

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/ptrnhist.c

DATE 5/23/89 PAGE #
TIME 6:14:48 pm 7/118

```

LINE # SOURCE TEXT
721 /* Result #4 */
722 (sampled_unk_ptr->word[word] & "sampled_data_ptr->word[word]) |
723
724 /* Result #1 */
725 ( (sampled_hiz_ptr->word[word] | sampled_unk_ptr->word[word]) &
726   sampled_data_ptr->word[word])
727
728 )
729
730 }
731
732 /* copy the control bits */
733 ((PTRN_BITS *)consistent_set_ptr->ctl =
734   ((PTRN_BITS *)ptrn_loaded_ptr->ctl,
735
736   ++ptrn_loaded_ptr,
737   ++sampled_data_ptr,
738   ++sampled_hiz_ptr,
739   ++sampled_unk_ptr,
740   ++sin_data_ptr,
741   ++sin_hiz_ptr,
742   ++sin_unk_ptr,
743   ++ident_inputs_ptr,
744   ++ident_outputs_ptr,
745   ++ident_ios_ptr,
746   ++consistent_set_ptr,
747
748 )
749
750 get_result(def_ptr, instance, ident_change,
751   event_type, eval_pin_number, eval_pin_count, any_driving_to_z)
752
753 DEVICE_SPEC def_ptr,
754 INSTANCE_INFO instance,
755 PTRN_BITS_LONGWORD ident_change,
756 u_char event_type,
757 u_short eval_pin_number,
758 u_short eval_pin_count,
759 u_char any_driving_to_z,
760
761 /* Read the result from the magic chip and set the variable "ident_change".
762   * "event_type" determines if it is legal to have IO STORE transitions.
763   * "eval_pin_number" and "eval_pin_count" are used to report errors,
764   * and ONLY used if "event_type" is EVAL_EVENT.
765   * Also change the output pin's instance->ptrn_loaded to be the new
766   * sample value.
767 */
768
769 DAB INFO dab_ptr,
770 EXTRA_DEVICE_SPEC extra_def_ptr,
771 PIN_SPEC pin_spec,
772 u_long new_sample_unk_ptr,
773 u_long new_sample_hiz_ptr,
774 u_long new_sample_data_ptr,
775 u_long last_sample_unk_ptr,
776 u_long last_sample_hiz_ptr,
777 u_long last_sample_data_ptr,
778 u_long ptrn_loaded_ptr,
779 u_long ident_ios_ptr,
780 u_long ident_outputs_ptr,
781 u_long ident_change_ptr,
782 u_long ident_store_ptr,
783 u_long ident_driving_to_z,
784 u_long ident_x_to_driving,
785 u_long ident_io_store,
786 u_long mask,
787 u_long ident_hiz_io_change,
788 u_short pin_number,
789 char pin_name_list[MAX_PIN_LENGTH],
790 u_char build_pin_name_list = FALSE,
791 u_char i,
792 u_char total_word,
793 u_char total_bits,
794 u_char waitno,
795 u_char wordno,
796 u_char bitno,
797 u_char unit_offset,
798 u_char word_offset,
799 u_short word_pin_number_offset,
800 u_short unit_pin_number_offset,
801
802 DPRINTF(("inside get_result\n"));
803
804 any_driving_to_z = FALSE;
805
806 extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
807 dab_ptr = dab_list[instance->dab_info_index];
808 total_word = dab_ptr->unit_count *
809   sizeof(PTRN_BITS_LONGWORD) / sizeof(u_long);
810
811 read_magic_full_sample_reg(instance, &gbl_new_sample_value);
812
813 new_sample_unk_ptr = (u_long *)&gbl_new_sample_value.unknown;
814 new_sample_hiz_ptr = (u_long *)&gbl_new_sample_value.hiz;
815 new_sample_data_ptr = (u_long *)&gbl_new_sample_value.data;
816
817 last_sample_unk_ptr = (u_long *)instance->last_sample_value.unknown;
818 last_sample_hiz_ptr = (u_long *)instance->last_sample_value.hiz;
819 last_sample_data_ptr = (u_long *)instance->last_sample_value.data;
820
821 ptrn_loaded_ptr = (u_long *)instance->ptrn_loaded;
822
823 ident_ios_ptr = (u_long *)extra_def_ptr->ident_ios;
824 ident_outputs_ptr = (u_long *)extra_def_ptr->ident_outputs;
825 ident_store_ptr = (u_long *)extra_def_ptr->ident_store;
826 ident_change_ptr = (u_long *)ident_change;
827
828 if (instance->first_eval == TRUE) {
829   DPRINTF(("first eval\n"));
830   for (wordno = 0; wordno < total_word; ++wordno) {
831     ident_change_ptr[wordno] |= 0xffffffff;
832
833     ident_change_ptr[wordno] |=
834       ident_ios_ptr[wordno] | ident_outputs_ptr[wordno];
835
836     ptrn_loaded_ptr[wordno] =
837       (ptrn_loaded_ptr[wordno] & "ident_ios_ptr[wordno]) |
838       (new_sample_data_ptr[wordno] & ident_ios_ptr[wordno]);
839
840 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/ptrnhist.c

*

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:48 pm | 8/119 |

```

LINE # SOURCE TEXT
841 instance->first_eval = FALSE;
842 }
843 else {
844     instance->first_eval = FALSE;
845     for (wordno = 0; wordno < total_word; ++wordno) {
846         /* Ignore the changes from I1 to I0 and vice versa */
847         ident_hiz_no_change =
848             new_sample_hiz_ptr[wordno] & last_sample_hiz_ptr[wordno];
849         ident_change_ptr[wordno] |=
850             new_sample_hiz_ptr[wordno] ^ last_sample_hiz_ptr[wordno];
851         new_sample_data_ptr[wordno] = last_sample_data_ptr[wordno];
852         new_sample_data_ptr[wordno] |=
853             ident_hiz_no_change;
854         /* We are interested only in IO and OUTPUT changes */
855         ident_change_ptr[wordno] |=
856             ident_ios_ptr[wordno] | ident_outputs_ptr[wordno];
857         DPRINTF(("chg word %d: %08x\n", wordno, ident_change_ptr[wordno]));
858         ptrs_loaded_ptr[wordno] =
859             (ptrs_loaded_ptr[wordno] & "ident_ios_ptr[wordno] | "
860              new_sample_data_ptr[wordno] & ident_ios_ptr[wordno]);
861     }
862     if (extra_def_ptr->has_io_store == TRUE) {
863         total_unit = def_ptr->unit_count;
864         /* Check if any IO STORE pins go from driving to 2 */
865         unit_offset = 0;
866         for (unitno = 0; unitno < total_unit; ++unitno) {
867             word_pin_number_offset = unit_pin_number_offset + 79;
868             for (wordno = 0; wordno < 3; ++wordno) {
869                 word_offset = unit_offset + wordno;
870                 ident_io_store = ident_ios_ptr[word_offset] &
871                     ident_store_ptr[word_offset];
872                 ptrs_loaded_ptr[word_offset] =
873                     (ptrs_loaded_ptr[word_offset] & "ident_io_store | "
874                     new_sample_data_ptr[word_offset] & ident_io_store);
875                 if (ident_driving_to_2 =
876                     (ident_io_store) &
877                     (last_sample_hiz_ptr[word_offset] &
878                     new_sample_hiz_ptr[word_offset])) {
879                     any_driving_to_2 = TRUE;
880                     for (bitno = 31; bitno >= 0; --bitno) {
881                         mask = bitno_to_mask(bitno);
882                         if (ident_driving_to_2 == 0)
883                             break;
884                         if (ident_driving_to_2 & mask) {
885                             if ((event_type == EVAL_EVENT) &&
886                                 (def_ptr->report_ioac == TRUE)) {
887                                 if (build_pin_name_list == FALSE) {
888                                     /* Get a list of EVAL pin names */
889                                     pin_name_list[0] = '\0';
890                                     for (i = 0; i < eval_pin_count; ++i) {
891                                         pin_spec = def_ptr->
892                                             pin_table[eval_pin_number[i]];
893                                         if (strlen(pin_name_list) +
894                                             strlen(pin_spec->pin_name) + 1 <
895                                             MAX_LINE_LENGTH) {
896                                             (void)strcat(pin_name_list,
897                                                 pin_spec->pin_name);
898                                             (void)strcat(pin_name_list, " ");
899                                         }
900                                         else
901                                             break;
902                                     }
903                                     build_pin_name_list = TRUE;
904                                 }
905                                 pin_number = word_pin_number_offset + bitno - 31;
906                                 in_query_message(WARNING_MSG, "any of these eval pins: %s of instance: %s caused I/O store pin: %s to change from driv"
907                                     ing to 2");
908                                 pin_name_list,
909                                 instance->device_info_string,
910                                 def_ptr->pin_table[pin_number].pin_name);
911                                 reset_ptr_bit_low(
912                                     (PTRN_BITS_LONGWORD *)&instance->hmdab_loaded[unitno],
913                                     (u_char)bitno);
914                             }
915                         }
916                     }
917                     word_pin_number_offset -= 32;
918                     unit_offset += sizeof(PTRN_BITS_LONGWORD) / sizeof(u_long);
919                     unit_pin_number_offset += 80;
920                 }
921                 /* Check if any IO STORE pins go from 2 to driving */
922                 unit_pin_number_offset = 0;

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/ptrnhist.c

DATE 5/23/89
TIME 6:14:48 pm

PAGE #
9/120

```

LINE #          SOURCE TEXT
960      unit_offset = 0;
961
962      for (unitno = 0; unitno < total_unit; ++unitno) {
963
964          word_pin_number_offset = unit_pin_number_offset + 79;
965
966          for (wordno = 0; wordno < 3; ++wordno) {
967
968              word_offset = unit_offset + wordno;
969
970              if (ident_x_to_driving =
971                  (ident_loc_ptr[word_offset] & ident_store_ptr[word_offset]) &
972                  (last_sample_hiz_ptr[word_offset] &
973                  new_sample_hiz_ptr[word_offset])) {
974
975                  instance->use_2_bit_per_pin = TRUE;
976
977                  for (bitno = 31; bitno >= 0; --bitno) {
978                      mask = bitno_to_mask(bitno);
979
980                      if (ident_x_to_driving == 0)
981                          break;
982
983                      if (ident_x_to_driving & mask) {
984                          ident_x_to_driving = mask;
985
986                          if ((event_type == EVAL_EVENT) &&
987                              (def_ptr->report_loc == TRUE)) {
988                              if (build_pin_name_list == FALSE) {
989
990                                  /* Get a list of EVAL pin names */
991                                  pin_name_list[0] = '\0';
992
993                                  for (i = 0; i < eval_pin_count; ++i) {
994                                      pin_spec = &def_ptr->
995                                          pin_table[eval_pin_number[i]];
996
997                                      if (strlen(pin_name_list) +
998                                          strlen(pin_spec->pin_name) <
999                                          MAX_LINE_LENGTH) {
1000                                          (void)strcat(pin_name_list,
1001                                              pin_spec->pin_name);
1002                                          (void)strcat(pin_name_list, " ");
1003                                      }
1004                                      else
1005                                          break;
1006                                  }
1007
1008                                  build_pin_name_list = TRUE;
1009                              }
1010
1011                                  pin_number = word_pin_number_offset + bitno - 31;
1012
1013                                  in_queue_message(WARNING_MSG, "any of these eval pins: %s of instance: %s caused I/O store pin: %s to change from 2 to
1014                                  driver",
1015                                  pin_name_list,
1016                                  instance->device_info_string,
1017                                  def_ptr->pin_table[pin_number].pin_name);
1018
1019                                  set_ptr->bit_long(
1020                                  (PTRN_BITS_LONGWORD *)&instance->hmdenb_loaded[unitno],
1021                                  wordno, (u_char)bitno);
1022
1023                                  }
1024
1025                                  }
1026
1027                                  word_pin_number_offset += 32;
1028
1029                                  unit_offset += sizeof(PTRN_BITS_LONGWORD) / sizeof(u_long);
1030                                  unit_pin_number_offset += 80;
1031
1032                                  }
1033
1034                                  /* Save new_sample_value to instance->last_sample_value */
1035                                  copy_ptr->bits((char *)&new_sample_val_ptr,
1036                                  (char *)&last_sample_val_ptr,
1037                                  &def_ptr->unit_count);
1038                                  copy_ptr->bits((char *)&new_sample_hiz_ptr,
1039                                  (char *)&last_sample_hiz_ptr,
1040                                  &def_ptr->unit_count);
1041                                  copy_ptr->bits((char *)&new_sample_data_ptr,
1042                                  (char *)&last_sample_data_ptr,
1043                                  &def_ptr->unit_count);
1044
1045                                  return(SUCCESS);
1046
1047
1048      check_consistency(instance, sample_result1, sample_result2,
1049      IDENT_INCONSISTENT_PINS, ident_change, total_unit)
1050
1051      INSTANCE_INFO      "instance:
1052      FULL_VALUE          "sample_result1,
1053      FULL_VALUE          "sample_result2,
1054      PTRN_BITS_LONGWORD  "ident_inconsistent_pins,
1055      PTRN_BITS_LONGWORD  "ident_change,
1056      u_char              total_unit,
1057
1058      /* Compare sample_result1 and sample_result2.
1059      * ADD the differences to ident_inconsistent_pins and ident_change.
1060      */
1061      EXTRA_DEVICE_SPEC  "extra_def_ptr:
1062      PTRN_BITS_LONGWORD  "s1_data_ptr,
1063      PTRN_BITS_LONGWORD  "s1_hiz_ptr,
1064      PTRN_BITS_LONGWORD  "s1_unknown_ptr,
1065      PTRN_BITS_LONGWORD  "s2_data_ptr,
1066      PTRN_BITS_LONGWORD  "s2_hiz_ptr,
1067      PTRN_BITS_LONGWORD  "s2_unknown_ptr,
1068      PTRN_BITS_LONGWORD  "ident_outputs_ptr,
1069      PTRN_BITS_LONGWORD  "ident_loc_ptr,
1070      u_char              unitno,
1071      u_char              wordno,
1072      #ifdef DEBUG
1073      u_long              inconsistent_pins,
1074      u_long              mask,
1075      u_short            pin_number,
1076      u_char              any_inconsistent_pins = FALSE,
1077      char               bitno.
1078      #endif

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/ptrnhist.c | DATE 5/23/89 | PAGE # 10/121 |
|--|--|-------------------------------------|-----------------|------------------|
| LINE # | SOURCE TEXT | | | |
| 1079 | extra_def_ptr = (EXTRA_DEVICE_SPEC *)instance->definition->extra_data, | | | |
| 1080 | ident_outputs_ptr = extra_def_ptr->ident_outputs, | | | |
| 1081 | ident_ios_ptr = extra_def_ptr->ident_ios, | | | |
| 1082 | s1_data_ptr = (PTRN_BITS_LONGWORD *)sample_result1->data, | | | |
| 1083 | s1_hiz_ptr = (PTRN_BITS_LONGWORD *)sample_result1->hiz, | | | |
| 1084 | s1_unknown_ptr = (PTRN_BITS_LONGWORD *)sample_result1->unknown, | | | |
| 1085 | s2_data_ptr = (PTRN_BITS_LONGWORD *)sample_result2->data, | | | |
| 1086 | s2_hiz_ptr = (PTRN_BITS_LONGWORD *)sample_result2->hiz, | | | |
| 1087 | s2_unknown_ptr = (PTRN_BITS_LONGWORD *)sample_result2->unknown, | | | |
| 1088 | for (unitno = 0; unitno < total_unit; ++unitno) { | | | |
| 1089 | for (wordno = 0; wordno < 3; ++wordno) { | | | |
| 1090 | ident_inconsistent_ptr = word[wordno]; | | | |
| 1091 | s1_data_ptr = word[wordno]; s2_data_ptr = word[wordno]; | | | |
| 1092 | s1_hiz_ptr = word[wordno]; s2_hiz_ptr = word[wordno]; | | | |
| 1093 | s1_unknown_ptr = word[wordno]; s2_unknown_ptr = word[wordno]; | | | |
| 1094 | ident_outputs_ptr = word[wordno]; | | | |
| 1095 | ident_ios_ptr = word[wordno]; | | | |
| 1096 | ident_change_ptr = word[wordno]; | | | |
| 1097 | ident_inconsistent_ptr = word[wordno]; | | | |
| 1098 | if (ident_inconsistent_ptr == 0) { | | | |
| 1099 | break; | | | |
| 1100 | mask = bitno_to_mask(bitno); | | | |
| 1101 | if (inconsistent_ptr & mask) { | | | |
| 1102 | inconsistent_ptr = mask; | | | |
| 1103 | pin_number = CALC_PIN_NUMBER_LONG(unitno, wordno, bitno); | | | |
| 1104 | DPRINTF(("PIN: %d is found inconsistent\n", pin_number)); | | | |
| 1105 | } | | | |
| 1106 | any_inconsistent_ptr = TRUE; | | | |
| 1107 | } | | | |
| 1108 | } | | | |
| 1109 | if (any_inconsistent_ptr == TRUE) { | | | |
| 1110 | DPRINTF(("there are inconsistent pins\n")); | | | |
| 1111 | } | | | |
| 1112 | } | | | |
| 1113 | check_consistency2(instance, sample_result1, sample_result2, | | | |
| 1114 | ident_inconsistent_ptr, ident_change, total_unit) | | | |
| 1115 | INSTANCE INFO | | | |
| 1116 | FULL VALUE | | | |
| 1117 | PTRN_BITS_LONGWORD | | | |
| 1118 | PTRN_BITS_LONGWORD | | | |
| 1119 | PTRN_BITS_LONGWORD | | | |
| 1120 | PTRN_BITS_LONGWORD | | | |
| 1121 | PTRN_BITS_LONGWORD | | | |
| 1122 | PTRN_BITS_LONGWORD | | | |
| 1123 | PTRN_BITS_LONGWORD | | | |
| 1124 | PTRN_BITS_LONGWORD | | | |
| 1125 | PTRN_BITS_LONGWORD | | | |
| 1126 | PTRN_BITS_LONGWORD | | | |
| 1127 | PTRN_BITS_LONGWORD | | | |
| 1128 | PTRN_BITS_LONGWORD | | | |
| 1129 | PTRN_BITS_LONGWORD | | | |
| 1130 | PTRN_BITS_LONGWORD | | | |
| 1131 | PTRN_BITS_LONGWORD | | | |
| 1132 | PTRN_BITS_LONGWORD | | | |
| 1133 | PTRN_BITS_LONGWORD | | | |
| 1134 | PTRN_BITS_LONGWORD | | | |
| 1135 | PTRN_BITS_LONGWORD | | | |
| 1136 | PTRN_BITS_LONGWORD | | | |
| 1137 | PTRN_BITS_LONGWORD | | | |
| 1138 | PTRN_BITS_LONGWORD | | | |
| 1139 | PTRN_BITS_LONGWORD | | | |
| 1140 | PTRN_BITS_LONGWORD | | | |
| 1141 | PTRN_BITS_LONGWORD | | | |
| 1142 | PTRN_BITS_LONGWORD | | | |
| 1143 | PTRN_BITS_LONGWORD | | | |
| 1144 | PTRN_BITS_LONGWORD | | | |
| 1145 | PTRN_BITS_LONGWORD | | | |
| 1146 | PTRN_BITS_LONGWORD | | | |
| 1147 | PTRN_BITS_LONGWORD | | | |
| 1148 | PTRN_BITS_LONGWORD | | | |
| 1149 | PTRN_BITS_LONGWORD | | | |
| 1150 | PTRN_BITS_LONGWORD | | | |
| 1151 | PTRN_BITS_LONGWORD | | | |
| 1152 | PTRN_BITS_LONGWORD | | | |
| 1153 | PTRN_BITS_LONGWORD | | | |
| 1154 | PTRN_BITS_LONGWORD | | | |
| 1155 | PTRN_BITS_LONGWORD | | | |
| 1156 | PTRN_BITS_LONGWORD | | | |
| 1157 | PTRN_BITS_LONGWORD | | | |
| 1158 | PTRN_BITS_LONGWORD | | | |
| 1159 | PTRN_BITS_LONGWORD | | | |
| 1160 | PTRN_BITS_LONGWORD | | | |
| 1161 | PTRN_BITS_LONGWORD | | | |
| 1162 | PTRN_BITS_LONGWORD | | | |
| 1163 | PTRN_BITS_LONGWORD | | | |
| 1164 | PTRN_BITS_LONGWORD | | | |
| 1165 | PTRN_BITS_LONGWORD | | | |
| 1166 | PTRN_BITS_LONGWORD | | | |
| 1167 | PTRN_BITS_LONGWORD | | | |
| 1168 | PTRN_BITS_LONGWORD | | | |
| 1169 | PTRN_BITS_LONGWORD | | | |
| 1170 | PTRN_BITS_LONGWORD | | | |
| 1171 | PTRN_BITS_LONGWORD | | | |
| 1172 | PTRN_BITS_LONGWORD | | | |
| 1173 | PTRN_BITS_LONGWORD | | | |
| 1174 | PTRN_BITS_LONGWORD | | | |
| 1175 | PTRN_BITS_LONGWORD | | | |
| 1176 | PTRN_BITS_LONGWORD | | | |
| 1177 | PTRN_BITS_LONGWORD | | | |
| 1178 | PTRN_BITS_LONGWORD | | | |
| 1179 | PTRN_BITS_LONGWORD | | | |
| 1180 | PTRN_BITS_LONGWORD | | | |
| 1181 | PTRN_BITS_LONGWORD | | | |
| 1182 | PTRN_BITS_LONGWORD | | | |
| 1183 | PTRN_BITS_LONGWORD | | | |
| 1184 | PTRN_BITS_LONGWORD | | | |
| 1185 | PTRN_BITS_LONGWORD | | | |
| 1186 | PTRN_BITS_LONGWORD | | | |
| 1187 | PTRN_BITS_LONGWORD | | | |
| 1188 | PTRN_BITS_LONGWORD | | | |
| 1189 | PTRN_BITS_LONGWORD | | | |
| 1190 | PTRN_BITS_LONGWORD | | | |
| 1191 | PTRN_BITS_LONGWORD | | | |
| 1192 | PTRN_BITS_LONGWORD | | | |
| 1193 | PTRN_BITS_LONGWORD | | | |
| 1194 | PTRN_BITS_LONGWORD | | | |
| 1195 | PTRN_BITS_LONGWORD | | | |
| 1196 | PTRN_BITS_LONGWORD | | | |
| 1197 | PTRN_BITS_LONGWORD | | | |
| 1198 | PTRN_BITS_LONGWORD | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/ptrnhist.c

DATE 5/23/89
TIME 6:14:48 pm

PAGE #
11/122

```

1199 ident_inconsistent_pins->word[wordno] |=
1200 (((s1_data_ptr ->word[wordno]) & s2_data_ptr ->word[wordno]) &
1201  (s1_hiz_ptr ->word[wordno] & s2_hiz_ptr ->word[wordno]) &
1202  (s1_unk_ptr ->word[wordno] & s2_unk_ptr ->word[wordno]) &
1203  (s1_outputs_ptr->word[wordno] &
1204   ident_ios_ptr->word[wordno]));
1205
1206 ident_change->word[wordno] |=
1207   ident_inconsistent_pins->word[wordno];
1208
1209 #ifdef DEBUG
1210 if (ident_inconsistent_pins->word[wordno]) {
1211   inconsistent_pins = ident_inconsistent_pins->word[wordno];
1212   for (bitno = 31; bitno >= 0; --bitno) {
1213     if (inconsistent_pins &= 0)
1214       break;
1215     mask = bitno <> mask[bitno];
1216     if (inconsistent_pins & mask) {
1217       inconsistent_pins = mask;
1218       pin_number = CALC_PIN_NUMBER_LONG(unitno, wordno, bitno);
1219       DPRINTF(("PN: %d is found inconsistent\n", pin_number));
1220     }
1221   }
1222   any_inconsistent_pins = TRUE;
1223 }
1224 #endif
1225
1226 ++ident_inconsistent_pins;
1227 ++ident_outputs_ptr;
1228 ++ident_ios_ptr;
1229 ++ident_change;
1230 ++s1_data_ptr;
1231 ++s1_hiz_ptr;
1232 ++s1_unk_ptr;
1233 ++s2_data_ptr;
1234 ++s2_hiz_ptr;
1235 ++s2_unk_ptr;
1236
1237 #ifdef DEBUG
1238 if (any_inconsistent_pins == TRUE) {
1239   DPRINTF(("there are inconsistent pins\n"));
1240 }
1241 #endif
1242
1243 copy_in_pin_changes(instance)
1244 INSTANCE_INFO *instance;
1245 {
1246   PIN_INFO *pin_info;
1247   u_short data_pin_count;
1248   u_short eval_pin_count;
1249   u_short store_pin_count;
1250   u_short i;
1251   u_short pin_number;
1252
1253   data_pin_count = lm_get_short();
1254   eval_pin_count = lm_get_short();
1255   store_pin_count = lm_get_short();
1256
1257   for (i = 0; i < data_pin_count; ++i) {
1258     pin_number = lm_get_short();
1259     pin_info = instance->pin_info_table(pin_number);
1260
1261     if (pin_info->input_pin_is_linked == TRUE) {
1262       lm_queue_message(ERROR_MSG, "internal error: duplicate pin info from HOST (pinno: %d)",
1263         pin_number);
1264       return(FAILURE);
1265     }
1266     else {
1267       pin_info->pin_time = lm_get_int();
1268       pin_info->new_row = lm_get_char();
1269
1270       /* Map 10/11 to 11 for pullup OR 10/11 to 10 for pulldown. */
1271       MAP_IN_LOGIC_VALUE(pin_number, pin_info->has_resistor,
1272         pin_info->new_row);
1273       pin_info->input_pin_is_linked = TRUE;
1274       pin_info->next_input_pin_index = instance->first_data_pin_index;
1275       instance->first_data_pin_index = pin_number;
1276     }
1277   }
1278
1279   for (i = 0; i < eval_pin_count; ++i) {
1280     pin_number = lm_get_short();
1281     pin_info = instance->pin_info_table(pin_number);
1282
1283     if (pin_info->input_pin_is_linked == TRUE) {
1284       lm_queue_message(ERROR_MSG, "internal error: duplicate pin info from HOST (pinno: %d)",
1285         pin_number);
1286       return(FAILURE);
1287     }
1288     else {
1289       pin_info->pin_time = lm_get_int();
1290       pin_info->new_row = lm_get_char();
1291
1292       /* Map 10/11 to 11 for pullup OR 10/11 to 10 for pulldown. */
1293       MAP_IN_LOGIC_VALUE(pin_number, pin_info->has_resistor,
1294         pin_info->new_row);
1295       pin_info->input_pin_is_linked = TRUE;
1296       pin_info->next_input_pin_index = instance->first_eval_pin_index;
1297       instance->first_eval_pin_index = pin_number;
1298     }
1299   }
1300
1301   for (i = 0; i < store_pin_count; ++i) {
1302     pin_number = lm_get_short();
1303     pin_info = instance->pin_info_table(pin_number);
1304
1305     if (pin_info->input_pin_is_linked == TRUE) {
1306       lm_queue_message(ERROR_MSG, "internal error: duplicate pin info from HOST (pinno: %d)",
1307         pin_number);
1308       return(FAILURE);
1309     }
1310     else {
1311

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/ptrnhist.c

DATE

5/23/89

PAGE #

12/123

TIME

6:14:48 pm

```

1319     pin_info->pin_time = lm_get_int();
1320     pin_info->new_raw = lm_get_char();
1321
1322     /* Map I0/I1 to I1 for pullup OR I0/I1 to I0 for pulldown. */
1323     MAP_IN_LOGIC_VALUE(pin_number, pin_info->has_resistor,
1324         pin_info->new_raw);
1325     pin_info->input_pin_is_linked = TRUE;
1326     pin_info->next_input_pin_index = instance->first_store_pin_index;
1327     instance->first_store_pin_index = pin_number;
1328
1329 }
1330
1331 return(SUCCESS);
1332 }
1333
1334 copy_out_pin_changes(instance)
1335 INSTANCE_INFO *instance;
1336 {
1337     /* Copy the IO and OUTPUT changes from the PIN_INFO TABLE to the
1338     * network buffer. These changes are linked with first_data_pin_index
1339     * as the head of the list.
1340     */
1341
1342     PIN_INFO *pin_info;
1343     u_long output_count_mark;
1344     short pin_number;
1345     short pin_count = 0;
1346
1347     output_count_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
1348     lm_put_short(0);
1349
1350     pin_number = instance->first_data_pin_index;
1351     instance->first_data_pin_index = -1;
1352
1353     while (pin_number != -1) {
1354         ++pin_count;
1355         pin_info = instance->pin_info_table[pin_number];
1356         pin_info->output_pin_is_linked = FALSE;
1357
1358         DPRINTF(("chg PW: ad val: td EPW: td ",
1359             pin_number, pin_info->old_filtered,
1360             pin_info->event_pin_number));
1361         lm_put_short(pin_number);
1362
1363         /* Map I1 to I1 for pullup or I0 to I0 for pulldown. */
1364         MAP_OUT_LOGIC_VALUE(pin_number, pin_info->has_resistor,
1365             pin_info->direction,
1366             pin_info->old_filtered);
1367
1368         lm_put_char(pin_info->old_filtered);
1369         lm_put_short(pin_info->event_pin_number);
1370         lm_put_char(pin_info->delay_type);
1371
1372         switch (pin_info->delay_type) {
1373             default:
1374                 case DELAY_TABLE_COMPOSITE:
1375                     DPRINTF(("td: td: td: ",
1376                         pin_info->min_delay,
1377                         pin_info->typ_delay, pin_info->max_delay));
1378                     lm_put_int(pin_info->min_delay);
1379                     lm_put_int(pin_info->typ_delay);
1380                     lm_put_int(pin_info->max_delay);
1381                     break;
1382                 case DELAY_TABLE:
1383                     DPRINTF(("td: td: ", pin_info->min_delay));
1384                     lm_put_int(pin_info->min_delay);
1385                     break;
1386                 case MEASURED:
1387                     DPRINTF(("meas: td-td: ", pin_info->min_delay, pin_info->max_delay));
1388                     lm_put_int(pin_info->min_delay);
1389                     lm_put_int(pin_info->max_delay);
1390                     break;
1391             }
1392         DPRINTF(("n="));
1393
1394         pin_number = pin_info->next_output_pin_index;
1395     }
1396
1397     LM_PUT_SHORT_AT_MARK(output_count_mark, lm_global_conn_ptr, pin_count);
1398 }
1399
1400 run_input_bcode(instance)
1401 INSTANCE_INFO *instance;
1402 {
1403     /* The B-code uses the pin values in OLD_RAW, OLD_FILTERED, NEW_RAW, and
1404     * NEW_FILTERED. And leave the new modified pin values in OLD_FILTERED.
1405     * It should maintain the pin change link list.
1406     */
1407
1408     /* Since we don't have the B-code yet, just copy the pin values from
1409     * NEW_RAW to OLD_FILTERED.
1410     */
1411
1412     PIN_INFO *pin_info;
1413     short pin_number;
1414
1415     /* Copy the DATA pin changes */
1416     pin_number = instance->first_data_pin_index;
1417
1418     while (pin_number != -1) {
1419         pin_info = instance->pin_info_table[pin_number];
1420         pin_info->old_filtered = pin_info->new_raw;
1421         pin_number = pin_info->next_input_pin_index;
1422     }
1423
1424     /* Copy the EVAL pin changes */
1425     pin_number = instance->first_eval_pin_index;
1426
1427     while (pin_number != -1) {
1428         pin_info = instance->pin_info_table[pin_number];
1429         pin_info->old_filtered = pin_info->new_raw;
1430         pin_number = pin_info->next_input_pin_index;
1431     }
1432
1433     /* Copy the STORE pin changes */
1434     pin_number = instance->first_store_pin_index;
1435
1436     while (pin_number != -1) {
1437         pin_info = instance->pin_info_table[pin_number];
1438         pin_info->old_filtered = pin_info->new_raw;
1439         pin_number = pin_info->next_input_pin_index;
1440     }
1441 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/ptrnhist.c

DATE 5/23/89
TIME 6:14:48 pm

PAGE #
13/124

```

1439
1440 while (pin_number != -1) {
1441     pin_info = instance->pin_info_table(pin_number);
1442     pin_info->old_filtered = pin_info->new_raw;
1443     pin_number = pin_info->next_input_pin_index;
1444 }
1445
1446 run_output_bcode(instance);
1447 INSTANCE_INFO *instance;
1448 {
1449     PIN_INFO *pin_info;
1450     short pin_number;
1451     /* Copy the OUTPUT pin changes */
1452     pin_number = instance->first_data_pin_index;
1453     while (pin_number != -1) {
1454         pin_info = instance->pin_info_table(pin_number);
1455         pin_info->old_filtered = pin_info->new_raw;
1456         pin_number = pin_info->next_output_pin_index;
1457     }
1458 }
1459
1460 copy_initial_values(instance);
1461 INSTANCE_INFO *instance;
1462 {
1463     DEVICE_SPEC *definition;
1464     UWB_OFFSET *uwb_ptr;
1465     PIN_SPEC *pin_ptr;
1466     PIN_INFO *pin_info;
1467     u_long output_pin_count_mark;
1468     u_short output_pin_count;
1469     u_short pinno;
1470     u_short pin_count;
1471     u_char pin_value;
1472     u_char unitno;
1473     u_char wordno;
1474     u_char bitno;
1475
1476     definition = instance->definition;
1477     output_pin_count_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
1478     lm_put_short(0); /* dummy out count */
1479     output_pin_count = 0;
1480     pin_count = definition->pin_cnt;
1481     pin_ptr = definition->pin_table;
1482     for (pinno = 0; pinno < pin_count; ++pinno) {
1483         switch (pin_ptr->direction) {
1484             case OUT:
1485                 case IO:
1486                     pin_info = instance->pin_info_table(pinno);
1487                     lm_put_short(pinno);
1488                     uwb_ptr = uwb_ptr->uwb_offset(pinno);
1489                     unitno = uwb_ptr->unitno;
1490                     wordno = uwb_ptr->wordno;
1491                     bitno = uwb_ptr->bitno;
1492
1493                     pin_value = read_pin_value(instance->last_sample_value,
1494                                             unitno, wordno, bitno);
1495
1496                     /* Map Z1 to Z1 for pullup or Z0 to Z0 for pulldown. */
1497                     MAP_OUT_LOGIC_VALUE(pinno, pin_info->has_resistor,
1498                                         pin_info->direction,
1499                                         pin_value);
1500
1501                     lm_put_char(pin_value);
1502                     DPRINTF(("initial val PN: %d val: %d\n", pinno, pin_value));
1503                     ++output_pin_count;
1504                     break;
1505
1506                     default:
1507                         break;
1508                 }
1509                 ++pin_ptr;
1510             }
1511
1512     LM_PUT_SHORT_AT_MARK(output_pin_count_mark,
1513                         lm_global_conn_ptr, output_pin_count);
1514 }
1515
1516 switch_to_2_bit_per_pin(instance);
1517 INSTANCE_INFO *instance;
1518 {
1519     /* Copy the patterns before the measurement patterns to the measurement
1520      * patterns and write the MEASURE patterns to the measurement patterns
1521      * address - 1.
1522      */
1523     DAB_INFO *dab_ptr;
1524     u_long *source_addr_ptr;
1525     u_long *dest_addr_ptr;
1526     u_long source_addr;
1527     u_long dest_addr;
1528     u_long temp;
1529     u_char unitno;
1530     u_char lane;
1531     u_char total_unit;
1532
1533     DPRINTF(("inside switch_to_2_bit_per_pin\n"));
1534
1535     #ifdef DONT_FIXIT
1536     write_patterns(instance,
1537                    instance->unit_addr(instance->cur_unit_addr_index)[0],
1538                    instance->hmem_loaded);
1539     #else
1540     source_addr_ptr = instance->unit_addr[
1541                    instance->cur_unit_addr_index + 1][0];
1542     dest_addr_ptr = instance->unit_addr(instance->cur_unit_addr_index)[0];
1543     dab_ptr = dab_list(instance->dab_info_index);
1544     total_unit = dab_ptr->lane_count - dab_ptr->unit_count_per_lane;
1545     for (unitno = 0; unitno < total_unit; ++unitno) {
1546         source_addr = source_addr_ptr[unitno];
1547     }
1548 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/ptrnhist.c

DATE 5/23/89
TIME 6:14:48 pm

PAGE #
14/125

| LINE # | SOURCE TEXT |
|--------|---|
| 1559 | dest_addr = dest_addr_ptr[unitno]; |
| 1560 | |
| 1561 | temp = read_loc_long((u_long *)source_addr); |
| 1562 | write_loc_long((u_long *)dest_addr, temp); |
| 1563 | |
| 1564 | temp = read_loc_long((u_long *) (source_addr + LANE_SEGMENT_B_OFFSET)); |
| 1565 | write_loc_long((u_long *) (dest_addr + LANE_SEGMENT_B_OFFSET), temp); |
| 1566 | |
| 1567 | temp = read_loc_long((u_long *) (source_addr + LANE_SEGMENT_C_OFFSET)); |
| 1568 | write_loc_long((u_long *) (dest_addr + LANE_SEGMENT_C_OFFSET), temp); |
| 1569 | |
| 1570 | } |
| 1571 | |
| 1572 | write_pattern(instance, |
| 1573 | (instance->unit_addr(instance->cur_unit_addr_index+1)[0], |
| 1574 | instance->hmdenb_loaded); |
| 1575 | endif |
| 1576 | |
| 1577 | if (grow_pattern(instance) == FAILURE) |
| 1578 | return(FAILURE); |
| 1579 | |
| 1580 | instance->use_2_bit_per_pin = FALSE; |
| 1581 | } |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/saverest.c

DATE 5/23/89
TIME 6:14:50 pm

PAGE #
1/126

```

LINE # SOURCE TEXT
1  /* SCOS_ID: saverest.c rev 1.1, 4/24/89 at 07:53:52 */
2
3  #include "device.h"
4  #include "hardware.h"
5  #include "aspram.h"
6  #include "lserver.h"
7  #include "lnetwork.h"
8  #include "network.h"
9
10 send_pattern(user, instance)
11 USER_INFO *user,
12 INSTANCE_INFO *instance,
13 {
14     /* Copy one device pattern to the output buffer.
15     * Return FALSE if there are no more patterns to copy to the buffer,
16     * otherwise return TRUE.
17     */
18
19     PTRN_BITS LONGWORD *ptrn_ptr;
20     DAB_INFO *dab_ptr;
21     u_long eddr;
22     u_long temp;
23     u_char unit_count;
24     u_char wordno;
25     u_char unitno;
26
27     dab_ptr = dab_list(instance->dab_info_index);
28
29     unit_count = dab_ptr->unit_count;
30
31     switch (user->save_state) {
32     case SENT_RECV_NOTHING:
33
34         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->ptrn_loaded[0];
35
36         for (unitno = 0; unitno < unit_count; ++unitno) {
37             for (wordno = 0; wordno < 3; ++wordno) {
38                 lm_put_int(ptrn_ptr->word[wordno]);
39             }
40             ++ptrn_ptr;
41         }
42
43         user->save_state = SENT_RECV_PTRN_LOADED;
44         break;
45     case SENT_RECV_PTRN_LOADED:
46
47         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->hmdenb_loaded[0];
48
49         for (unitno = 0; unitno < unit_count; ++unitno) {
50             for (wordno = 0; wordno < 3; ++wordno) {
51                 lm_put_int(ptrn_ptr->word[wordno]);
52             }
53             ++ptrn_ptr;
54         }
55
56         user->save_state = SENT_RECV_HMDENB_LOADED;
57         break;
58     case SENT_RECV_HMDENB_LOADED:
59
60         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->lcychdb_loaded[0];
61
62         for (unitno = 0; unitno < unit_count; ++unitno) {
63             for (wordno = 0; wordno < 3; ++wordno) {
64                 lm_put_int(ptrn_ptr->word[wordno]);
65             }
66             ++ptrn_ptr;
67         }
68
69         user->save_state = SENT_RECV_LCYCHDB_LOADED;
70         break;
71     case SENT_RECV_LCYCHDB_LOADED:
72
73         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->lcycmdb_loaded[0];
74
75         for (unitno = 0; unitno < unit_count; ++unitno) {
76             for (wordno = 0; wordno < 3; ++wordno) {
77                 lm_put_int(ptrn_ptr->word[wordno]);
78             }
79             ++ptrn_ptr;
80         }
81
82         user->save_state = SENT_RECV_LCYCMB_LOADED;
83         break;
84     case SENT_RECV_LCYCMB_LOADED:
85
86         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->last_consistent_set[0];
87
88         for (unitno = 0; unitno < unit_count; ++unitno) {
89             for (wordno = 0; wordno < 3; ++wordno) {
90                 lm_put_int(ptrn_ptr->word[wordno]);
91             }
92             ++ptrn_ptr;
93         }
94
95         user->save_state = SENT_RECV_LAST_CONSISTENT_SET;
96         break;
97     case SENT_RECV_LAST_CONSISTENT_SET:
98
99         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.data[0];
100
101         for (unitno = 0; unitno < unit_count; ++unitno) {
102             for (wordno = 0; wordno < 3; ++wordno) {
103                 lm_put_int(ptrn_ptr->word[wordno]);
104             }
105             ++ptrn_ptr;
106         }
107
108         user->save_state = SENT_RECV_SIM_PIN_VALUE_DATA;
109         break;
110     case SENT_RECV_SIM_PIN_VALUE_DATA:
111
112         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.hiz[0];
113
114         for (unitno = 0; unitno < unit_count; ++unitno) {
115             for (wordno = 0; wordno < 3; ++wordno) {
116                 lm_put_int(ptrn_ptr->word[wordno]);
117             }
118             ++ptrn_ptr;
119         }
120

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/saverest.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:50 pm | 2/127 |

| LINE # | SOURCE TEXT |
|--------|--|
| 121 | user->save_state = SENT_RECV_SIM_PIN_VALUE_HIZ; |
| 122 | break; |
| 123 | case SENT_RECV_SIM_PIN_VALUE_HIZ: |
| 124 | ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.unknown[0]; |
| 125 | |
| 126 | for (unitno = 0; unitno < unit_count; ++unitno) { |
| 127 | for (wordno = 0; wordno < 3; ++wordno) { |
| 128 | lm_put_int(ptrn_ptr->word[wordno]); |
| 129 | |
| 130 | ++ptrn_ptr; |
| 131 | |
| 132 | } |
| 133 | |
| 134 | user->save_state = SENT_RECV_SIM_PIN_VALUE_UNK; |
| 135 | break; |
| 136 | case SENT_RECV_SIM_PIN_VALUE_UNK: |
| 137 | |
| 138 | ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.soft[0]; |
| 139 | |
| 140 | for (unitno = 0; unitno < unit_count; ++unitno) { |
| 141 | for (wordno = 0; wordno < 3; ++wordno) { |
| 142 | lm_put_int(ptrn_ptr->word[wordno]); |
| 143 | |
| 144 | ++ptrn_ptr; |
| 145 | |
| 146 | } |
| 147 | |
| 148 | user->save_state = SENT_RECV_SIM_PIN_VALUE_SOFT; |
| 149 | break; |
| 150 | case SENT_RECV_SIM_PIN_VALUE_SOFT: |
| 151 | |
| 152 | ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.data[0]; |
| 153 | |
| 154 | for (unitno = 0; unitno < unit_count; ++unitno) { |
| 155 | for (wordno = 0; wordno < 3; ++wordno) { |
| 156 | lm_put_int(ptrn_ptr->word[wordno]); |
| 157 | |
| 158 | ++ptrn_ptr; |
| 159 | |
| 160 | } |
| 161 | |
| 162 | user->save_state = SENT_RECV_LAST_SAMPLE_VALUE_DATA; |
| 163 | break; |
| 164 | case SENT_RECV_LAST_SAMPLE_VALUE_DATA: |
| 165 | |
| 166 | ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.hiz[0]; |
| 167 | |
| 168 | for (unitno = 0; unitno < unit_count; ++unitno) { |
| 169 | for (wordno = 0; wordno < 3; ++wordno) { |
| 170 | lm_put_int(ptrn_ptr->word[wordno]); |
| 171 | |
| 172 | ++ptrn_ptr; |
| 173 | |
| 174 | } |
| 175 | |
| 176 | user->save_state = SENT_RECV_LAST_SAMPLE_VALUE_HIZ; |
| 177 | break; |
| 178 | case SENT_RECV_LAST_SAMPLE_VALUE_HIZ: |
| 179 | |
| 180 | ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.unknown[0]; |
| 181 | |
| 182 | for (unitno = 0; unitno < unit_count; ++unitno) { |
| 183 | for (wordno = 0; wordno < 3; ++wordno) { |
| 184 | lm_put_int(ptrn_ptr->word[wordno]); |
| 185 | |
| 186 | ++ptrn_ptr; |
| 187 | |
| 188 | } |
| 189 | |
| 190 | user->save_state = SENT_RECV_LAST_SAMPLE_VALUE_UNK; |
| 191 | break; |
| 192 | case SENT_RECV_LAST_SAMPLE_VALUE_UNK: |
| 193 | |
| 194 | ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.soft[0]; |
| 195 | |
| 196 | for (unitno = 0; unitno < unit_count; ++unitno) { |
| 197 | for (wordno = 0; wordno < 3; ++wordno) { |
| 198 | lm_put_int(ptrn_ptr->word[wordno]); |
| 199 | |
| 200 | ++ptrn_ptr; |
| 201 | |
| 202 | } |
| 203 | |
| 204 | user->save_state = SENT_RECV_LAST_SAMPLE_VALUE_SOFT; |
| 205 | break; |
| 206 | case SENT_RECV_LAST_SAMPLE_VALUE_SOFT: |
| 207 | |
| 208 | if (user->pattern_to_send_count == 0) { |
| 209 | user->save_state = SENT_RECV_PATTERN; |
| 210 | return(FALSE); |
| 211 | } |
| 212 | |
| 213 | for (unitno = 0; unitno < unit_count; ++unitno) { |
| 214 | user->last_unit_sent_addr[unitno] = |
| 215 | instance->first_user_ptrn_unit_addr[unitno]; |
| 216 | |
| 217 | addr = user->last_unit_sent_addr[unitno]; |
| 218 | |
| 219 | temp = read_loc_long((u_long *)addr); |
| 220 | |
| 221 | lm_put_int(temp); |
| 222 | |
| 223 | temp = read_loc_long((u_long *) (addr + LANE_SEGMENT_B_OFFSET)); |
| 224 | |
| 225 | lm_r = int(temp); |
| 226 | |
| 227 | temp = read_loc_long((u_long *) (addr + LANE_SEGMENT_C_OFFSET)); |
| 228 | |
| 229 | lm_put_int(temp); |
| 230 | |
| 231 | } |
| 232 | |
| 233 | increment_unit_addr(user->last_unit_sent_addr, |
| 234 | unit_count, dab_ptr->unit_count_per_lane); |
| 235 | |
| 236 | user->pattern_to_send_count -= dab_ptr->unit_count_per_lane; |
| 237 | |
| 238 | user->save_state = SENT_RECV_PATTERN; |
| 239 | break; |
| 240 | case SENT_RECV_PATTERN: |
| 241 | |
| 242 | if (user->pattern_to_send_count == 0) { |
| 243 | return(FALSE); |
| 244 | } |
| 245 | |
| 246 | for (unitno = 0; unitno < unit_count; ++unitno) { |
| 247 | addr = user->last_unit_sent_addr[unitno]; |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/saverest.c

DATE 5/23/89
TIME 6:14:50 pm

PAGE #
3/128

LINE # SOURCE TEXT

```

241     temp = read_loc_long((u_long *)addr);
242
243     lm_put_int(temp);
244
245     temp = read_loc_long((u_long *)(&addr + LANE_SEGMENT_B_OFFSET));
246
247     lm_put_int(temp);
248
249     temp = read_loc_long((u_long *)(&addr + LANE_SEGMENT_C_OFFSET));
250
251     lm_put_int(temp);
252
253     }
254
255     increment_unit_addr(user->last_unit_sent_addr, unit_count,
256                        dab_ptr->unit_count_per_lane);
257
258     user->pattern_to_send_count -- dab_ptr->unit_count_per_lane;
259
260     break;
261
262     default:
263         return(FALSE);
264     }
265
266     return(TRUE);
267 }
268
269 restore_inst_pattern(instance, unit_count)
270 INSTANCE_INFO *instance;
271 u_short unit_count;
272 {
273     DAB_INFO *dab_ptr;
274     PTRN_BITS_LONGWORD *ptrn_ptr;
275     PTRN_BITS temp_ptrn[MAX_UNIT_COUNT];
276     u_char unitno;
277     u_char wordno;
278
279     dab_ptr = dab_list(instance->dab_info_index);
280
281     switch (instance->restore_state) {
282     case SENT_RECV_NOTHING:
283         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->ptrn_loaded[0];
284
285         for (unitno = 0; unitno < unit_count; ++unitno) {
286             for (wordno = 0; wordno < 3; ++wordno) {
287                 ptrn_ptr->word[wordno] = lm_get_int();
288             }
289
290             fix_pel_ctl_word(ptrn_ptr, unitno, dab_ptr);
291
292             ++ptrn_ptr;
293         }
294
295         instance->restore_state = SENT_RECV_PTRN_LOADED;
296         break;
297     case SENT_RECV_PTRN_LOADED:
298         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->hdenb_loaded[0];
299
300         for (unitno = 0; unitno < unit_count; ++unitno) {
301             for (wordno = 0; wordno < 3; ++wordno) {
302                 ptrn_ptr->word[wordno] = lm_get_int();
303             }
304
305             fix_pel_ctl_word(ptrn_ptr, unitno, dab_ptr);
306
307             ++ptrn_ptr;
308         }
309
310         instance->restore_state = SENT_RECV_HDENB_LOADED;
311         break;
312     case SENT_RECV_HDENB_LOADED:
313         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->lcychdb_loaded[0];
314
315         for (unitno = 0; unitno < unit_count; ++unitno) {
316             for (wordno = 0; wordno < 3; ++wordno) {
317                 ptrn_ptr->word[wordno] = lm_get_int();
318             }
319
320             fix_pel_ctl_word(ptrn_ptr, unitno, dab_ptr);
321
322             ++ptrn_ptr;
323         }
324
325         instance->restore_state = SENT_RECV_LCYCHDB_LOADED;
326         break;
327     case SENT_RECV_LCYCHDB_LOADED:
328         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->lcycmdb_loaded[0];
329
330         for (unitno = 0; unitno < unit_count; ++unitno) {
331             for (wordno = 0; wordno < 3; ++wordno) {
332                 ptrn_ptr->word[wordno] = lm_get_int();
333             }
334
335             fix_pel_ctl_word(ptrn_ptr, unitno, dab_ptr);
336
337             ++ptrn_ptr;
338         }
339
340         instance->restore_state = SENT_RECV_LCYCMB_LOADED;
341         break;
342     case SENT_RECV_LCYCMB_LOADED:
343         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->last_consistent_set[0];
344
345         for (unitno = 0; unitno < unit_count; ++unitno) {
346             for (wordno = 0; wordno < 3; ++wordno) {
347                 ptrn_ptr->word[wordno] = lm_get_int();
348             }
349
350             fix_pel_ctl_word(ptrn_ptr, unitno, dab_ptr);
351
352             ++ptrn_ptr;
353         }
354
355         instance->restore_state = SENT_RECV_LAST_CONSISTENT_SET;
356         break;
357     case SENT_RECV_LAST_CONSISTENT_SET:
358         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.data[0];
359
360         for (unitno = 0; unitno < unit_count; ++unitno) {

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/saverest.c

DATE 5/23/89

PAGE #

TIME 6:14:50 pm

4/129

| LINE # | SOURCE TEXT |
|--------|--|
| 361 | for (wordao = 0; wordao < 3; ++wordao) { |
| 362 | ptrs_ptr->word[wordao] = lm_get_int(); |
| 363 | } |
| 364 | ++ptrs_ptr; |
| 365 | } |
| 366 | instance->restore_state = SENT_RECV_SIM_PIN_VALUE_DATA; |
| 367 | break; |
| 368 | case SENT_RECV_SIM_PIN_VALUE_DATA: |
| 369 | ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.hiz[0]; |
| 370 | for (unitao = 0; unitao < unit_count; ++unitao) { |
| 371 | for (wordao = 0; wordao < 3; ++wordao) { |
| 372 | ptrs_ptr->word[wordao] = lm_get_int(); |
| 373 | } |
| 374 | ++ptrs_ptr; |
| 375 | } |
| 376 | instance->restore_state = SENT_RECV_SIM_PIN_VALUE_HIZ; |
| 377 | break; |
| 378 | case SENT_RECV_SIM_PIN_VALUE_HIZ: |
| 379 | ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.unknown[0]; |
| 380 | for (unitao = 0; unitao < unit_count; ++unitao) { |
| 381 | for (wordao = 0; wordao < 3; ++wordao) { |
| 382 | ptrs_ptr->word[wordao] = lm_get_int(); |
| 383 | } |
| 384 | ++ptrs_ptr; |
| 385 | } |
| 386 | instance->restore_state = SENT_RECV_SIM_PIN_VALUE_UNK; |
| 387 | break; |
| 388 | case SENT_RECV_SIM_PIN_VALUE_UNK: |
| 389 | ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.soft[0]; |
| 390 | for (unitao = 0; unitao < unit_count; ++unitao) { |
| 391 | for (wordao = 0; wordao < 3; ++wordao) { |
| 392 | ptrs_ptr->word[wordao] = lm_get_int(); |
| 393 | } |
| 394 | ++ptrs_ptr; |
| 395 | } |
| 396 | instance->restore_state = SENT_RECV_SIM_PIN_VALUE_SOFT; |
| 397 | break; |
| 398 | case SENT_RECV_SIM_PIN_VALUE_SOFT: |
| 399 | ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.data[0]; |
| 400 | for (unitao = 0; unitao < unit_count; ++unitao) { |
| 401 | for (wordao = 0; wordao < 3; ++wordao) { |
| 402 | ptrs_ptr->word[wordao] = lm_get_int(); |
| 403 | } |
| 404 | ++ptrs_ptr; |
| 405 | } |
| 406 | instance->restore_state = SENT_RECV_SIM_PIN_VALUE_DATA; |
| 407 | break; |
| 408 | case SENT_RECV_SIM_PIN_VALUE_DATA: |
| 409 | ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.hiz[0]; |
| 410 | for (unitao = 0; unitao < unit_count; ++unitao) { |
| 411 | for (wordao = 0; wordao < 3; ++wordao) { |
| 412 | ptrs_ptr->word[wordao] = lm_get_int(); |
| 413 | } |
| 414 | ++ptrs_ptr; |
| 415 | } |
| 416 | instance->restore_state = SENT_RECV_LAST_SAMPLE_VALUE_DATA; |
| 417 | break; |
| 418 | case SENT_RECV_LAST_SAMPLE_VALUE_DATA: |
| 419 | ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.hiz[0]; |
| 420 | for (unitao = 0; unitao < unit_count; ++unitao) { |
| 421 | for (wordao = 0; wordao < 3; ++wordao) { |
| 422 | ptrs_ptr->word[wordao] = lm_get_int(); |
| 423 | } |
| 424 | ++ptrs_ptr; |
| 425 | } |
| 426 | instance->restore_state = SENT_RECV_LAST_SAMPLE_VALUE_HIZ; |
| 427 | break; |
| 428 | case SENT_RECV_LAST_SAMPLE_VALUE_HIZ: |
| 429 | ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.unknown[0]; |
| 430 | for (unitao = 0; unitao < unit_count; ++unitao) { |
| 431 | for (wordao = 0; wordao < 3; ++wordao) { |
| 432 | ptrs_ptr->word[wordao] = lm_get_int(); |
| 433 | } |
| 434 | ++ptrs_ptr; |
| 435 | } |
| 436 | instance->restore_state = SENT_RECV_LAST_SAMPLE_VALUE_UNK; |
| 437 | break; |
| 438 | case SENT_RECV_LAST_SAMPLE_VALUE_UNK: |
| 439 | ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.soft[0]; |
| 440 | for (unitao = 0; unitao < unit_count; ++unitao) { |
| 441 | for (wordao = 0; wordao < 3; ++wordao) { |
| 442 | ptrs_ptr->word[wordao] = lm_get_int(); |
| 443 | } |
| 444 | ++ptrs_ptr; |
| 445 | } |
| 446 | instance->restore_state = SENT_RECV_LAST_SAMPLE_VALUE_SOFT; |
| 447 | break; |
| 448 | case SENT_RECV_LAST_SAMPLE_VALUE_SOFT: |
| 449 | ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.soft[0]; |
| 450 | for (unitao = 0; unitao < unit_count; ++unitao) { |
| 451 | for (wordao = 0; wordao < 3; ++wordao) { |
| 452 | ptrs_ptr->word[wordao] = lm_get_int(); |
| 453 | } |
| 454 | ++ptrs_ptr; |
| 455 | } |
| 456 | fix_pel_ctl_word(ptrs_ptr, unitao, dab_ptr); |
| 457 | ++ptrs_ptr; |
| 458 | } |
| 459 | write_patterns(instance, |
| 460 | instance->unit_addr[instance->cur_unit_addr_index][0], |
| 461 | (PTRN_BITS *)instance->last_sample_value.soft[0]; |
| 462 | if (grow_patterns(instance) == FAILURE) |
| 463 | return(FAILURE); |
| 464 | instance->restore_state = SENT_RECV_PATTERN; |
| 465 | break; |
| 466 | case SENT_RECV_PATTERN: |
| 467 | ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.soft[0]; |
| 468 | for (unitao = 0; unitao < unit_count; ++unitao) { |
| 469 | for (wordao = 0; wordao < 3; ++wordao) { |
| 470 | ptrs_ptr->word[wordao] = lm_get_int(); |
| 471 | } |
| 472 | ++ptrs_ptr; |
| 473 | } |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/saverest.c

DATE

5/23/89

PAGE #

TIME

6:14:50 pm

5/130

LINE # SOURCE TEXT

```
481     }
482     fix_pel_ctl_word(ptra_ptr, unitso, dab_ptr),
483     484
485     ++ptrb, ---
486     }
487     write_pattern(instance,
488     489     {instance->unit_addr[instance->cur_unit_addr_index][0],
490     491     (PTRN_BITS * )temp_ptrn[0]);
492     if (grow_pattern(instance) == FAILURE)
493     494     return(FAILURE);
495     break;
496     default:
497     498     break;
499     }
500     return(SUCCESS);
501 }
```

1583

5,353,243

1584

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/timer.c

DATE 5/23/89
TIME 6:14:50 pm

PAGE #
1/131

| LINE # | SOURCE TEXT |
|--------|---|
| 1 | /* SCCS ID: timer.c rev 3.1, 4/24/89 at 07:53:56 */ |
| 2 | |
| 3 | #ifndef MODELER |
| 4 | #include "gval.h" |
| 5 | #include "s/time.h" |
| 6 | #include "common.h" |
| 7 | |
| 8 | static u_long lm_tick = 0; |
| 9 | |
| 10 | static |
| 11 | handle_SIGALRM() |
| 12 | { |
| 13 | ++lm_tick; |
| 14 | } |
| 15 | |
| 16 | static struct itimerval initial_value = { { 0, 20000 }, { 0, 20000 } }; |
| 17 | |
| 18 | inittimer() |
| 19 | { |
| 20 | struct itimerval oldval; |
| 21 | signal(SIGVTALRM, handle_SIGALRM); |
| 22 | setitimer(ITIMER_VIRTUAL, &initial_value, &oldval); |
| 23 | } |
| 24 | |
| 25 | lm_time() |
| 26 | { |
| 27 | return(lm_tick); |
| 28 | } |
| 29 | |
| 30 | lm_delay(delay) |
| 31 | { |
| 32 | /* "delay" is msec */ |
| 33 | u_long start; |
| 34 | u_long tick; |
| 35 | tick = delay / 20 + 1; |
| 36 | start = lm_time(); |
| 37 | while ((lm_time() - start) < tick) |
| 38 | ; |
| 39 | } |
| 40 | |
| 41 | #endif |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM
lm1000/tmeas.c

DATE 5/23/89

PAGE #

TIME 6:14:50 pm

1/132

```

LINE # SOURCE TEXT
1  /* SCCS_ID: tmeas.c rev 1.4, 5/9/89 at 17:35:09 */
2
3  #include "device.h"
4  #include "message.h"
5  #include "hardware.h"
6  #include "soprom.h"
7  #include "laserware.h"
8  #include "protana.h"
9
10 #define MAGIC_OIN_OOUT_DELAY 400
11 #define MAGIC_OIN_IOUT_DELAY 1200
12 #define MAGIC_IIN_OOUT_DELAY -1000
13 #define MAGIC_IIN_IOUT_DELAY -1000
14
15 #define MAX_PIN_NAME_LIST 128
16
17 #ifdef DEBUG
18 #define SWEEP_RANGE 0
19 #else
20 #define SWEEP_RANGE 0
21 #endif
22
23 #define TRY_COUNT 1
24
25 /* The following is used to accumulate all changes from eval/store pins */
26 PTRN_BITS LONGWORD gbl_tm_ident_change[MAX_UNIT_COUNT];
27 PTRN_BITS LONGWORD gbl_tm_ident_inconsistent_pins[MAX_UNIT_COUNT];
28 PTRN_BITS gbl_tm_ident_data_change[MAX_UNIT_COUNT];
29
30 #ifdef DEBUG
31 extern u_char loop_till_key;
32 extern u_long debug_key;
33 extern u_long debug_char;
34 #endif
35
36 /* ABCUSED */
37 tm_evaluate_1_bit_per_pin(uchar, def_ptr, instance,
38                          ident_inconsistent_pins,
39                          steady_state_result,
40                          temp_steady_state_result,
41                          ident_change,
42                          changed_dac)
43
44 USER_INFO *user;
45 DEVICE_SPEC *def_ptr;
46 INSTANCE_INFO *instance;
47 PTRN_BITS LONGWORD *ident_inconsistent_pins;
48 FULL_VALUE *steady_state_result;
49 FULL_VALUE *temp_steady_state_result;
50 PTRN_BITS LONGWORD *ident_change;
51 u_char *changed_dac;
52
53 PIN_INFO *pin_info;
54 EXTRA_DEVICE_SPEC *extra_def_ptr;
55 PTRN_BITS consistent_set[MAX_UNIT_COUNT];
56 PTRN_BITS next_ptrn[MAX_UNIT_COUNT];
57 PTRN_BITS lcyddb[MAX_UNIT_COUNT];
58 PTRN_BITS lcyddb(MAX_UNIT_COUNT);
59 PTRN_BITS LONGWORD *lcyddb_ptr;
60 PTRN_BITS LONGWORD *lcyddb_ptr;
61 PTRN_BITS LONGWORD *ident_ios_ptr;
62 PTRN_BITS LONGWORD *inst_lcyddb_ptr;
63 PTRN_BITS LONGWORD *inst_lcyddb_ptr;
64 DAB_INFO *dab_ptr;
65 PIN_SPEC *pin_def;
66 UMS_OFFSET *ums_ptr;
67 u_long inst_block_number[MAX_LANE_COUNT];
68 u_long next_ptrn_addr[MAX_UNIT_COUNT];
69 LANE_ADDR_INFO last_unit_addr[MAX_LANE_COUNT];
70 u_long seq_end_addr[MAX_LANE_COUNT];
71 u_long timeout;
72 short pin_number;
73 u_short eval_index = 0;
74 u_char pin_value;
75 u_char old_pin_value;
76 u_char allocated_blocks;
77 u_char junk1;
78 u_char junk2;
79 u_char total_unit;
80 u_char unitno;
81 u_char wordno;
82 u_char bitno;
83 u_char evaluation_count = 0;
84 u_char old_val;
85 u_char i;
86 u_char any_driving_to_x;
87 u_char measure;
88
89 DPRINTF(("inside tm_evaluate_1_bit_per_pin\n"));
90
91 /*-----
92 * The timeout value is set with the assumption that we are running at
93 * 150KHz (1/6.66 usec period).
94 * timeout = pattern play time + 1 second for feedback
95 *          = pattern_count * 8 microsec + 125000 * 8 microsec
96 *          = (pattern_count + 125000) * 8 microsec
97 *          = (pattern_count + 125000) * 8 / 1000 millisecc
98 *          = (pattern_count + 125000) >> 7;
99 *-----
100 timeout = (instance->pattern_count + PTRN_COUNT_FUDGE_FACTOR) >> 7;
101
102 extra_def_ptr = (EXTRA_DEVICE_SPEC *) def_ptr->extra_data;
103 dab_ptr = dab_list(instance->dab_info_index);
104 total_unit = dab_ptr->unit_count;
105
106 clear_ptrn_bits((char *) gbl_tm_ident_change, total_unit);
107 clear_ptrn_bits((char *) gbl_tm_ident_inconsistent_pins, total_unit);
108 clear_ptrn_bits((char *) gbl_tm_ident_data_change, total_unit);
109
110 /*
111 * We don't need to write the LUTCHDB and LUTCHDB patterns here as the
112 * evaluate_1_bit_per_pin() because these will always be written below.
113 */
114
115 init_tm_pin_info_table(gbl_tm_pin_info_table, def_ptr->pin_cnt);
116
117 if (get_next_pattern_addr(instance, next_ptrn_addr,
118                          last_unit_addr, &allocated_blocks) == FAILURE)
119     return (FAILURE);
120

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/tmeas.c

DATE 5/23/89 PAGE #
TIME 6:14:50 pm 2/133

```

121 calculate_consistent_set(instance, def_ptr, setup_ptr);
122
123 /* Soft drive I/O pins during measurement pattern */
124 lcyddb_ptr = (PTRN_BITS_LONGWORD *) lcyddb;
125 lcyddb_ptr = (PTRN_BITS_LONGWORD *) lcyddb;
126 inst_lcyddb_ptr = (PTRN_BITS_LONGWORD *) instance->lcyddb_loaded;
127 inst_lcyddb_ptr = (PTRN_BITS_LONGWORD *) instance->lcyddb_loaded;
128 ident_ios_ptr = extra_def_ptr->ident_ios;
129
130 for (unitno = 0; unitno < total_unit; ++unitno) {
131     for (wordno = 0; wordno < 3; ++wordno) {
132
133         lcyddb_ptr->word[wordno] =
134             inst_lcyddb_ptr->word[wordno] | ident_ios_ptr->word[wordno];
135
136         lcyddb_ptr->word[wordno] =
137             inst_lcyddb_ptr->word[wordno] | ident_ios_ptr->word[wordno];
138     }
139
140     ++lcyddb_ptr;
141     ++lcyddb_ptr;
142     ++inst_lcyddb_ptr;
143     ++inst_lcyddb_ptr;
144     ++ident_ios_ptr;
145 }
146
147 pin_number = instance->first_data_pin_index;
148 instance->first_data_pin_index = -1;
149
150 while (pin_number != -1) {
151
152     web_ptr = sps_to_short_offset(pin_number);
153     unitno = web_ptr->unitno;
154     wordno = web_ptr->wordno;
155     bitno = web_ptr->bitno;
156
157     set_ptrn_bit(&tbl_tm_ident_data_change(unitno, wordno, bitno));
158
159     pin_def = &def_ptr->pin_table(pin_number);
160     pin_info = &instance->pin_info_table(pin_number);
161     pin_info->input_pin_is_linked = FALSE;
162     pin_value = pin_info->old_filtered;
163
164     /* Keep track of pin changes to build normal pattern sequence. */
165     set_pin_value(&instance->pin_value,
166                 unitno, wordno, bitno, pin_value);
167
168     set_measurement_pattern(instance, pin_def,
169                             unitno, wordno, bitno, pin_value,
170                             &junk1, &junk2);
171
172     set_pattern(instance, pin_def,
173                 setup_ptr, unitno, wordno, bitno, pin_value);
174
175     pin_number = pin_info->next_input_pin_index;
176 }
177
178 /* Write the setup pattern */
179 write_pattern(instance,
180               &instance->unit_addr[instance->cur_unit_addr_index][0],
181               setup_ptr);
182
183 /* Process the eval pins */
184 for (pin_number = instance->first_eval_pin_index;
185     pin_number != -1;
186     pin_number = pin_info->next_input_pin_index) {
187
188     /* Note that eval pin can only have ENRZ format. */
189
190     web_ptr = sps_to_short_offset(pin_number);
191     unitno = web_ptr->unitno;
192     wordno = web_ptr->wordno;
193     bitno = web_ptr->bitno;
194
195     pin_def = &def_ptr->pin_table(pin_number);
196     pin_info = &instance->pin_info_table(pin_number);
197     pin_info->input_pin_is_linked = FALSE;
198     pin_value = pin_info->old_filtered;
199
200     /* Keep track of pin changes to build normal pattern sequence. */
201     set_pin_value(&instance->pin_value,
202                 unitno, wordno, bitno, pin_value);
203
204     set_measurement_pattern(instance, pin_def,
205                             unitno, wordno, bitno, pin_value,
206                             &junk1, &junk2);
207
208     old_val = read_ptrn_bit(&setup_ptr[unitno], wordno, bitno);
209
210     set_pattern(instance, pin_def,
211                 setup_ptr, unitno, wordno, bitno, pin_value);
212
213     if (old_val == read_ptrn_bit(&setup_ptr[unitno], wordno, bitno)) {
214
215         /*
216          * If the pin value in the last consistent set is equal to the pin
217          * value in measurement pattern, then we cannot measure this
218          * transition.
219          */
220         DPRINTF(("cannot measure eval pin: %d\n", pin_number));
221         continue;
222     }
223
224     if (eval_index < MAX_EVAL_CHANGES) {
225         gbl_eval_pin_number[eval_index] = (u_short) pin_number;
226         gbl_eval_pin_value[eval_index++] = pin_value;
227     } else {
228         lm_queue_message(WARNING_MSG, "internal error: not enough room to store eval changes, delay number might be incorrect");
229     }
230
231     if (pin_def->direction == IO) {
232         reset_ptrn_bit(&lcyddb[unitno], wordno, bitno);
233         reset_ptrn_bit(&lcyddb[unitno], wordno, bitno);
234     }
235 }
236
237 if (eval_index != 0) {
238     /* Evaluation complete */
239     write_pattern(instance, &instance->lcyddb_addr[0], lcyddb);
240     write_pattern(instance, &instance->lcyddb_addr[0], lcyddb);

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/tmeas.c

DATE 5/23/89
TIME 6:14:50 pm

PAGE #
3/134

```

LINE # SOURCE TEXT
241 next_off_output(instance, setup_ptr);
242 write_pattern(instance, next_ptr_addr, setup_ptr);
243
244
245 set_seq_end_bit(instance,
246     last_unit_addr,
247     allocated_blocks,
248     seq_end_addr,
249     inst_block_number);
250
251 if (measure_delay(def_ptr, instance, timeout,
252     EVAL_EVENT, gbl_eval_pin_number, eval_index,
253     steady_state_result,
254     ident_inconsistent_pins,
255     temp_steady_state_result,
256     gbl_pin_info_table,
257     ident_change, changed_dac) == FAILURE) {
258     remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
259
260     if (allocated_blocks == TRUE)
261         return_last_blocks(instance);
262
263     return (FAILURE);
264 }
265 remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
266
267 /* Soft drive the IO EVAL pins for next evaluation */
268 for (i = 0; i < eval_index; ++i) {
269     pin_number = gbl_eval_pin_number[i];
270     if (def_ptr->pin_table[pin_number].direction == IO) {
271
272         uwb_ptr = uwb_ptr + short_offset[pin_number];
273         unitno = uwb_ptr->unitno;
274         wordno = uwb_ptr->wordno;
275         bitno = uwb_ptr->bitno;
276
277         set_ptrn_bit(&lcychdb[unitno], wordno, bitno);
278         set_ptrn_bit(&lcychdb[unitno], wordno, bitno);
279     }
280 }
281
282 /* Restore the sequence to normal state. */
283 write_pattern(instance,
284     instance->lcychdb_addr[0],
285     instance->lcychdb_loaded);
286 write_pattern(instance,
287     instance->lcychdb_addr[0],
288     instance->lcychdb_loaded);
289 write_pattern(instance,
290     instance->unit_addr[instance->cur_unit_addr_index][0],
291     instance->ptrn_loaded);
292
293 /*
294  * Run normal pattern history and check that the result obtained is the
295  * same as with timing measurement pattern history.
296  */
297
298 set_seq_end_bit(instance,
299     instance->last_addr,
300     FALSE,
301     seq_end_addr,
302     inst_block_number);
303
304
305 if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
306     remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
307
308     if (allocated_blocks == TRUE)
309         return_last_blocks(instance);
310
311     return (FAILURE);
312 }
313 if (get_result(def_ptr, instance, ident_change,
314     EVAL_EVENT, gbl_eval_pin_number,
315     eval_index, last_driving_to_1) == FAILURE) {
316     remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
317
318     if (allocated_blocks == TRUE)
319         return_last_blocks(instance);
320
321     return (FAILURE);
322 }
323 check_consistency2(instance,
324     instance->last_sample_value,
325     steady_state_result,
326     ident_inconsistent_pins,
327     ident_change,
328     total_unit);
329
330 for (i = 0; i < eval_index; ++i) {
331     pin_number = gbl_eval_pin_number[i];
332     if (def_ptr->pin_table[pin_number].direction == IO) {
333
334         uwb_ptr = uwb_ptr + short_offset[pin_number];
335         unitno = uwb_ptr->unitno;
336         wordno = uwb_ptr->wordno;
337         bitno = uwb_ptr->bitno;
338
339         reset_ptrn_bit(&ident_inconsistent_pins[unitno], wordno, bitno);
340
341         /*
342          * For timing measurement the eval pins will be sampled as driving
343          * (because we turn on LCYCHDB) and sampled as 1 for regular
344          * pattern history, and therefore the ident_change will be set for
345          * these eval pins. These are bogus changes so we need to reset the
346          * ident_change bit for these pins.
347          */
348         reset_ptrn_bit(&ident_change[unitno], wordno, bitno);
349     }
350 }
351
352 for (i = 0; i < eval_index; ++i) {
353
354     /*
355      * Get the table delays for all pins. Some of the delays will be
356      * overridden later if it turned out that the pin delays can be
357      * measured. This get_delay() loop has to be after we reset the
358      * ident_change bit for the eval pins because otherwise we would see
359      * unspc delays from one eval pin to another eval pin.
360      */

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/tmeas.c

DATE 5/23/89
TIME 6:14:50 pm

PAGE #
4/135

```

LINE # SOURCE TEXT
361 get_delay(def_ptr, instance, ident_change,
362 ident_inconsistent_pin,
363 gbl_eval_pin_number[i], gbl_eval_pin_value[i]);
364 }
365
366 /* Accumulate the pin changes in gbl_tm_ident_change
367 accumulate_ident_pins(ident_change, gbl_tm_ident_change, total_unit);
368
369 copy_ptrn_bits((char *) ident_inconsistent_pins,
370 (char *) gbl_tm_ident_inconsistent_pins,
371 total_unit);
372
373 add_inconsistent_pins(instance, ident_inconsistent_pins);
374
375 clear_ptrn_bits((char *) ident_change, total_unit);
376 clear_ptrn_bits((char *) ident_inconsistent_pins, total_unit);
377
378 if (any_driving_to_x == TRUE) {
379
380 /*
381 * Doe 2 bits per pin --> hard drive those JO STORE pins which change
382 * from driving to x.
383 */
384
385 if (def_ptr->device_type == PUBLIC) {
386
387 /*
388 * Write ADDRESS LOCKED to previous pattern address. For PRIVATE
389 * devices we don't have to do anything since ADDRESS will always be
390 * written when we do the next evaluation.
391 */
392 write_pattern(instance,
393 instance->unit_addr[
394 instance->cur_unit_addr_index + 1 & 1][0],
395 instance->hndwb_loaded);
396
397 write_pattern(instance,
398 instance->unit_addr[instance->cur_unit_addr_index][0],
399 instance->last_consistent_set);
400
401 if (grow_pattern(instance) == FAILURE)
402 return (FAILURE);
403
404 }
405
406 remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
407
408 if (allocated_blocks == TRUE)
409 return_last_blocks(instance);
410
411 /* Process the STORE pins */
412 for (pin_number = instance->first_store_pin_index,
413 instance->first_store_pin_index = -1,
414 pin_number != -1,
415 pin_info = pin_info->next_input_pin_index) {
416
417 gbl_eval_pin_number[0] = (u_short) pin_number;
418
419 uwb_ptr = &pin_to_short_offset[pin_number];
420 unitno = uwb_ptr->unitno;
421 wordno = uwb_ptr->wordno;
422 bitno = uwb_ptr->bitno;
423
424 pin_info = instance->pin_info_table[pin_number];
425 pin_def = def_ptr->pin_table[pin_number];
426
427 pin_info->input_pin_is_locked = FALSE;
428 pin_value = pin_info->old_filtered;
429
430 if (pin_def->direction == IN) {
431 /* INPUT STORE change */
432
433 old_pin_value = read_pin_value(instance->pin_pin_value,
434 unitno, wordno, bitno);
435
436 set_pin_value(instance->pin_pin_value,
437 unitno, wordno, bitno, pin_value);
438
439 switch (pin_def->clk_format) {
440 case RIZ:
441 case RIZI:
442 if (input_pin_transition(old_pin_value, pin_value,
443 pin_info->uninitialized_pin) == NO_TRANSITION) {
444
445 /*
446 * This is an error because the host should have filtered the
447 * transition. On second thought it is NOT an error because
448 * the host only filters transitions to exactly the same
449 * value.
450 */
451 DPRINTF(("No transition on IN STORE RIZ pin %s",
452 pin_def->pin_name));
453 continue;
454
455 if (pin_value & (LOGIC_0 | LOGIC_50 | LOGIC_10))
456 reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
457 else
458 set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
459 break;
460
461 case RI:
462 if (input_pin_transition(old_pin_value, pin_value,
463 pin_info->uninitialized_pin) != RISE_TRANSITION) {
464 DPRINTF(("No transition on IN STORE RI pin %s",
465 pin_def->pin_name));
466 continue;
467
468 /* Disable the RI clock on the measurement pattern (ptrn_loaded) */
469 set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
470 break;
471
472 case RO:
473 if (input_pin_transition(old_pin_value, pin_value,
474 pin_info->uninitialized_pin) != FALL_TRANSITION) {
475 DPRINTF(("No transition on IN STORE RO pin %s",
476 pin_def->pin_name));
477 continue;
478
479

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/tmeas.c

DATE 5/23/89

PAGE #

TIME 6:14:50 pm

5/136

```

LINE # SOURCE TEXT
481      /* Disable the R1 clock on the measurement pattern (ptrn_loaded) */
482      reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
483
484      break;
485      default:
486          in_queue_message(ERROR_MSG, "internal error: illegal pin type in device.h");
487          continue;
488      }
489  } else {
490      /* IO STORE change */
491      /* Note: IO store pin can only have DMEZ format */
492
493      old_pin_value = read_pin_value(instance->last_sample_value,
494                                   unitno, wordno, bitno);
495
496      set_pin_value(instance->pin_value,
497                   unitno, wordno, bitno, pin_value);
498
499      if (io_pin_transition(old_pin_value, pin_value,
500                           pin_info->uninitialized_pin) == NO_TRANSITION) {
501          DPRINTF(("no transition on IO STORE pin %s", pin_def->pin_name));
502          continue;
503      }
504      set_measurement_pattern(instance, pin_def,
505                             unitno, wordno, bitno, pin_value,
506                             &junk1, &junk2);
507  }
508
509  if (instance->max_2_bit_per_pin) {
510      switch_to_2_bit_per_pin(instance);
511  }
512
513  measure = TRUE;
514  switch (pin_def->clk_format) {
515      case R1:
516          write_pattern(instance,
517                       &instance->unit_addr[instance->cur_unit_addr_index][0],
518                       setup_ptrn);
519
520          if (grow_pattern(instance) == FAILURE)
521              return (FAILURE);
522
523          reset_ptrn_bit(setup_ptrn[unitno], wordno, bitno);
524          break;
525      case R0:
526          write_pattern(instance,
527                       &instance->unit_addr[instance->cur_unit_addr_index][0],
528                       setup_ptrn);
529
530          if (grow_pattern(instance) == FAILURE)
531              return (FAILURE);
532
533          set_ptrn_bit(setup_ptrn[unitno], wordno, bitno);
534          break;
535      default:
536          old_val = read_ptrn_bit(setup_ptrn[unitno], wordno, bitno);
537
538          write_pattern(instance,
539                       &instance->unit_addr[instance->cur_unit_addr_index][0],
540                       setup_ptrn);
541
542          set_pattern(instance, pin_def,
543                     setup_ptrn, unitno, wordno, bitno, pin_value);
544
545          if (old_val == read_ptrn_bit(setup_ptrn[unitno], wordno, bitno)) {
546              DPRINTF(("cannot measure store PM: %d\n", pin_number));
547              measure = FALSE;
548              if (grow_pattern(instance) == FAILURE)
549                  return (FAILURE);
550          }
551          break;
552      }
553
554  if (measure == TRUE) {
555      /*evaluation_count;
556
557      write_pattern(instance, &instance->lcychdb_addr[0], lcychdb);
558      write_pattern(instance, &instance->lcychdb_addr[0], lcychdb);
559      mask_off_output(instance, setup_ptrn);
560      write_pattern(instance,
561                   &instance->unit_addr[instance->cur_unit_addr_index][0],
562                   setup_ptrn);
563
564      if (measure == TRUE) {
565          set_seq_end_bit(instance,
566                        instance->seq_addr,
567                        FALSE,
568                        seq_end_addr,
569                        inst_block_number);
570
571          if (measure_delay(def_ptr, instance, timeout,
572                           STORE_EVENT, (u_short *) &pin_number, 1,
573                           steady_state_result,
574                           ident_inconsistent_pin,
575                           temp_steady_state_result,
576                           gbl_pin_info_table,
577                           ident_change, changed_dac) == FAILURE) {
578              remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
579              return (FAILURE);
580          }
581          remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
582      }
583      /* Sort drive this IO pin for next evaluation */
584      if (pin_def->direction == IO) {
585          set_ptrn_bit(&lcychdb[unitno], wordno, bitno);
586          set_ptrn_bit(&lcychdb[unitno], wordno, bitno);
587      }
588      /* Make the pattern history look like normal pattern history */
589      calculate_consistent_set(instance, def_ptr, consistent_set);
590      switch (pin_def->clk_format) {

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/tmeas.c

DATE 5/23/89
TIME 6:14:50 pm
PAGE # 6/137

```

LINE # SOURCE TEXT
601 case R1:
602     reset_ptrn_bit(consistent_set(unitno), wordno, bitno);
603     /* Disable the R1 clock on the setup ptrn so it will be
604     * correct when we evaluation the next store change in
605     * this simulation pass.
606     */
607     set_ptrn_bit(consistent_set(unitno), wordno, bitno);
608     break;
609 case R0:
610     set_ptrn_bit(consistent_set(unitno), wordno, bitno);
611     /* Disable the R0 clock on the setup ptrn so it will be
612     * correct when we evaluation the next store change in
613     * this simulation pass.
614     */
615     reset_ptrn_bit(consistent_set(unitno), wordno, bitno);
616     break;
617 default:
618     break;
619 }
620
621 /* save the consistent set */
622 copy_ptrn_bits((char *) consistent_set,
623               (char *) instance->last_consistent_set,
624               dab_list(instance)->dab_info_index->unit_count);
625
626 write_pattern(instance, instance->lcycmbb_addr[0],
627              instance->lcycmbb_loaded);
628
629 write_pattern(instance, instance->lcycmbb_addr[0],
630              instance->lcycmbb_loaded);
631
632 write_pattern(instance,
633              instance->
634              unit_addr(instance->cur_unit_addr_index + 1 & 1)[0],
635              consistent_set);
636
637 write_pattern(instance,
638              instance->unit_addr(instance->cur_unit_addr_index)[0],
639              instance->ptrn_loaded);
640
641 set_seq_end_bit(instance,
642               instance->seq_end_addr,
643               FALSE,
644               seq_end_addr,
645               inst_block_number);
646
647
648 if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
649     remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
650     return (FAILURE);
651 }
652 if (get_result(def_ptr, instance, ident_change,
653              STORE_EVENT, gbl_eval_pin_number,
654              1, any_driving_to_x) == FAILURE) {
655     remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
656     return (FAILURE);
657 }
658 if (measure == TRUE) {
659     check_consistency(instance,
660                     instance->last_sample_value,
661                     steady_state_result,
662                     ident_inconsistent_pins,
663                     ident_change,
664                     total_unit);
665 }
666 reset_ptrn_bit(instance_inconsistent_pins(unitno), wordno, bitno);
667 reset_ptrn_bit(instance_inconsistent_pins(unitno), wordno, bitno);
668
669 get_delay(def_ptr, instance, ident_change,
670          instance_inconsistent_pins,
671          (u_short) pin_number, pin_value);
672
673 /* Accumulate the pin changes in gbl_tm_ident_change */
674 accumulate_ident_pins(instance_inconsistent_pins, gbl_tm_ident_change, total_unit);
675
676 copy_ptrn_bits((char *) ident_inconsistent_pins,
677               (char *) gbl_tm_ident_inconsistent_pins,
678               total_unit);
679
680 add_inconsistent_pins(instance, ident_inconsistent_pins);
681
682 clear_ptrn_bits((char *) ident_change, total_unit);
683 clear_ptrn_bits((char *) ident_inconsistent_pins, total_unit);
684
685 if (any_driving_to_x == TRUE) {
686     /*
687     * Use 2 bits per pin --> hard drive those IO STORE pins which change
688     * from driving to 1.
689     */
690
691     if (def_ptr->device_type == PUBLIC) {
692         /*
693         * Write HMDENB_LOADED to previous pattern address. For PRIVATE
694         * devices we don't have to do anything since HMDENB will always be
695         * written when we do the next evaluation.
696         */
697         write_pattern(instance,
698                     instance->unit_addr[
699                     instance->cur_unit_addr_index + 1 & 1][0],
700                     instance->hmdenb_loaded);
701
702         write_pattern(instance,
703                     instance->unit_addr(instance->cur_unit_addr_index)[0],
704                     instance->last_consistent_set);
705
706         if (grow_pattern(instance) == FAILURE)
707             return (FAILURE);
708     }
709     remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
710
711     pin_info->uninitialized_pin = FALSE;
712 }
713
714 if (evaluation_count == 0)
715     return (SUCCESS);
716
717 copy_tm_result(instance, gbl_tm_pin_info_table,

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/tmeas.c

DATE 5/23/89
TIME 6:14:50 pm

PAGE #
7/138

```

LINE #          SOURCE TEXT
721          gbl_tm_ident_change, gbl_tm_ident_inconsistent_pins);
722
723      return (SUCCESS);
724  }
725
726  set_ptrn(instance, pin_def, ptrn_ptr, unitno, wordno, bitno, pin_value)
727  INSTANCE_INFO *instance;
728  PIN_SPEC      *pin_def;
729  PTRN_BITS     *ptrn_ptr;
730  u_char        unitno;
731  u_char        wordno;
732  u_char        bitno;
733  u_char        pin_value;
734  {
735      u_char      last_sample_value;
736
737      if (pin_def->direction == IN) {
738          if (pin_value & (LOGIC_0 | LOGIC_50))
739              reset_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
740          else
741              set_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
742      } else {
743          /* IO pin --> let the DWT pin decide. */
744          last_sample_value =
745              read_pin_value(instance->last_sample_value, unitno, wordno, bitno);
746
747          if (last_sample_value & LOGIC_0)
748              reset_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
749          else if (last_sample_value & LOGIC_1)
750              set_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
751          else if (last_sample_value & (LOGIC_10 | LOGIC_11)) {
752              if (pin_value & (LOGIC_0 | LOGIC_50))
753                  reset_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
754              else if (pin_value & (LOGIC_1 | LOGIC_51))
755                  set_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
756              else if (last_sample_value & LOGIC_10)
757                  reset_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
758              else
759                  set_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
760          } else {
761              /* Last sample is U */
762              toggle_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
763          }
764      }
765      last_sample_value =
766          read_pin_value(instance->last_sample_value, unitno, wordno, bitno);
767      copy_tm_result(instance, tm_pin_info_table, ident_change,
768                    ident_inconsistent_pins);
769      INSTANCE_INFO *instance;
770      TM_PIN_INFO   *tm_pin_info_table;
771      PTRN_BITS_LONGWORD *ident_change;
772      PTRN_BITS_LONGWORD *ident_inconsistent_pins;
773  {
774      DAB_INFO      *dab_ptr;
775      TM_PIN_INFO   *tm_pin_info;
776      PIN_INFO      *pin_info;
777      u_long        mask;
778      u_long        ident_change_word;
779      u_short        unit_pin_number_offset;
780      u_short        word_pin_number_offset;
781      u_short        dest_pin_number;
782      u_char         dest_pin_value;
783      u_char         total_unit;
784      u_char         unitno;
785      u_char         wordno;
786      char           bitno;
787
788      dab_ptr = dab_list(instance->dab_info_index);
789      total_unit = dab_ptr->unit_count;
790      unit_pin_number_offset = 0;
791      for (unitno = 0; unitno < total_unit; ++unitno) {
792          word_pin_number_offset = unit_pin_number_offset + 79;
793
794          for (wordno = 0; wordno < 3; ++wordno) {
795              ident_change_word = ident_change[unitno].word[wordno];
796
797              for (bitno = 31; bitno >= 0; --bitno) {
798                  if (ident_change_word == 0)
799                      break;
800
801                  mask = bitno_to_mask(bitno);
802                  if (ident_change_word & mask) {
803                      /* Reset the bit */
804                      ident_change_word = mask;
805
806                      dest_pin_number = word_pin_number_offset + bitno - 31;
807                      tm_pin_info = tm_pin_info_table[dest_pin_number];
808                      pin_info = instance->pin_info_table[dest_pin_number];
809
810                      if (read_ptrn_bit_long(ident_inconsistent_pins[unitno],
811                                           wordno, (u_char) bitno) != 0) {
812                          dest_pin_value = LOGIC_U;
813                          tm_pin_info->delay_value_is_set = FALSE;
814                      } else {
815                          dest_pin_value =
816                              read_pin_value(instance->last_sample_value,
817                                              unitno, wordno, (u_char) bitno);
818                      }
819
820                      pin_info->new_val = dest_pin_value;
821                      if (tm_pin_info->delay_value_is_set == TRUE) {
822                          /* Return measured delay */
823                          DPRINTF(("measured PN: %d val: %d delay: %d-%d\n",
824                                  dest_pin_number,
825                                  dest_pin_value,
826                                  tm_pin_info->min_delay,
827                                  tm_pin_info->max_delay));
828
829                          if (pin_info->output_pin_is_linked == FALSE) {
830                              pin_info->next_output_pin_index =
831                                  instance->first_data_pin_index;
832                              instance->first_data_pin_index = dest_pin_number;
833                          }
834                      }
835                  }
836              }
837          }
838      }
839  }
840

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/tmeas.c | DATE 5/23/89 | PAGE # 8/139 |
|--|--|---|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 841 | | pin_info->output_pin_is_linked = TRUE; | | |
| 842 | | pin_info->event_pin_number = | | |
| 843 | | tm_pin_info->event_pin_number; | | |
| 844 | | pin_info->delay_type = MEASURED; | | |
| 845 | | | | |
| 846 | | pin_info->min_delay = tm_pin_info->min_delay; | | |
| 847 | | pin_info->max_delay = tm_pin_info->max_delay; | | |
| 848 | | } else { | | |
| 849 | | /* pin delay was not measured */ | | |
| 850 | | PRINTF(("Not measured PM: id val: %d\n", | | |
| 851 | | dest_pin_number, | | |
| 852 | | dest_pin_value)); | | |
| 853 | | } | | |
| 854 | | } | | |
| 855 | | } | | |
| 856 | | word_pin_number_offset -- 32; | | |
| 857 | | } | | |
| 858 | | | | |
| 859 | | unit_pin_number_offset -- 80; | | |
| 860 | | } | | |
| 861 | | | | |
| 862 | | | | |
| 863 | | get_next_pattern_addr(instance, new_unit_addr, | | |
| 864 | | last_unit_addr, allocated_blocks) | | |
| 865 | | INSTANCE_INFO *instance; | | |
| 866 | | u_long new_unit_addr; | | |
| 867 | | LANE_ADDR_INFO *last_unit_addr; | | |
| 868 | | u_char *allocated_blocks; | | |
| 869 | | { | | |
| 870 | | /* ??? */ | | |
| 871 | | DAB_INFO *dab_ptr; | | |
| 872 | | LANE_ADDR_INFO *lane_info; | | |
| 873 | | u_long new_block_addr[MAX_LANE_COUNT]; | | |
| 874 | | u_long cur_unit_addr; | | |
| 875 | | addr_inc_per_lane; | | |
| 876 | | u_char lane_no; | | |
| 877 | | u_char unitno; | | |
| 878 | | u_char total_unit; | | |
| 879 | | u_char junk; | | |
| 880 | | { | | |
| 881 | | dab_ptr = dab_list(instance->dab_info_index); | | |
| 882 | | addr_inc_per_lane = dab_ptr->unit_count_per_lane * PTRN_ADDR_INC; | | |
| 883 | | total_unit = dab_ptr->lane_count * dab_ptr->unit_count_per_lane; | | |
| 884 | | | | |
| 885 | | *allocated_blocks = FALSE; | | |
| 886 | | for (lane_no = 0; lane_no < MAX_LANE_COUNT; ++lane_no) | | |
| 887 | | new_block_addr[lane_no] = 0; | | |
| 888 | | | | |
| 889 | | cur_unit_addr = *instance->unit_addr(instance->cur_unit_addr_index)[0]; | | |
| 890 | | for (unitno = 0; unitno < total_unit; ++unitno) { | | |
| 891 | | lane_no = dab_ptr->unit_location[unitno].lane_no; | | |
| 892 | | lane_info = *instance->lane_addr(lane_no); | | |
| 893 | | if ((cur_unit_addr[unitno] + addr_inc_per_lane) >= | | |
| 894 | | lane_info->max_addr) { | | |
| 895 | | if (new_block_addr[lane_no] == 0) { | | |
| 896 | | if (new_block_addr[lane_no] == 0) { | | |
| 897 | | if (new_block_addr[lane_no] == 0) { | | |
| 898 | | if (new_block_addr[lane_no] == 0) { | | |
| 899 | | if (new_block_addr[lane_no] == 0) { | | |
| 900 | | if (new_block_addr[lane_no] == 0) { | | |
| 901 | | if (new_block_addr[lane_no] == 0) { | | |
| 902 | | if (new_block_addr[lane_no] == 0) { | | |
| 903 | | if (new_block_addr[lane_no] == 0) { | | |
| 904 | | if (new_block_addr[lane_no] == 0) { | | |
| 905 | | if (new_block_addr[lane_no] == 0) { | | |
| 906 | | if (new_block_addr[lane_no] == 0) { | | |
| 907 | | if (new_block_addr[lane_no] == 0) { | | |
| 908 | | if (new_block_addr[lane_no] == 0) { | | |
| 909 | | if (new_block_addr[lane_no] == 0) { | | |
| 910 | | if (new_block_addr[lane_no] == 0) { | | |
| 911 | | if (new_block_addr[lane_no] == 0) { | | |
| 912 | | if (new_block_addr[lane_no] == 0) { | | |
| 913 | | if (new_block_addr[lane_no] == 0) { | | |
| 914 | | if (new_block_addr[lane_no] == 0) { | | |
| 915 | | if (new_block_addr[lane_no] == 0) { | | |
| 916 | | if (new_block_addr[lane_no] == 0) { | | |
| 917 | | if (new_block_addr[lane_no] == 0) { | | |
| 918 | | if (new_block_addr[lane_no] == 0) { | | |
| 919 | | if (new_block_addr[lane_no] == 0) { | | |
| 920 | | if (new_block_addr[lane_no] == 0) { | | |
| 921 | | if (new_block_addr[lane_no] == 0) { | | |
| 922 | | if (new_block_addr[lane_no] == 0) { | | |
| 923 | | if (new_block_addr[lane_no] == 0) { | | |
| 924 | | if (new_block_addr[lane_no] == 0) { | | |
| 925 | | if (new_block_addr[lane_no] == 0) { | | |
| 926 | | if (new_block_addr[lane_no] == 0) { | | |
| 927 | | if (new_block_addr[lane_no] == 0) { | | |
| 928 | | if (new_block_addr[lane_no] == 0) { | | |
| 929 | | if (new_block_addr[lane_no] == 0) { | | |
| 930 | | if (new_block_addr[lane_no] == 0) { | | |
| 931 | | if (new_block_addr[lane_no] == 0) { | | |
| 932 | | if (new_block_addr[lane_no] == 0) { | | |
| 933 | | if (new_block_addr[lane_no] == 0) { | | |
| 934 | | if (new_block_addr[lane_no] == 0) { | | |
| 935 | | if (new_block_addr[lane_no] == 0) { | | |
| 936 | | if (new_block_addr[lane_no] == 0) { | | |
| 937 | | if (new_block_addr[lane_no] == 0) { | | |
| 938 | | if (new_block_addr[lane_no] == 0) { | | |
| 939 | | if (new_block_addr[lane_no] == 0) { | | |
| 940 | | if (new_block_addr[lane_no] == 0) { | | |
| 941 | | if (new_block_addr[lane_no] == 0) { | | |
| 942 | | if (new_block_addr[lane_no] == 0) { | | |
| 943 | | if (new_block_addr[lane_no] == 0) { | | |
| 944 | | if (new_block_addr[lane_no] == 0) { | | |
| 945 | | if (new_block_addr[lane_no] == 0) { | | |
| 946 | | if (new_block_addr[lane_no] == 0) { | | |
| 947 | | if (new_block_addr[lane_no] == 0) { | | |
| 948 | | if (new_block_addr[lane_no] == 0) { | | |
| 949 | | if (new_block_addr[lane_no] == 0) { | | |
| 950 | | if (new_block_addr[lane_no] == 0) { | | |
| 951 | | if (new_block_addr[lane_no] == 0) { | | |
| 952 | | if (new_block_addr[lane_no] == 0) { | | |
| 953 | | if (new_block_addr[lane_no] == 0) { | | |
| 954 | | if (new_block_addr[lane_no] == 0) { | | |
| 955 | | measure_delay(def_ptr, instance, timeout, | | |
| 956 | | event_type, event_pin_number, event_pin_count, | | |
| 957 | | steady_state_result, ident_inconsistent_pins, | | |
| 958 | | temp_steady_state_result, tm_pin_info_table, | | |
| 959 | | ident_change, changed_dec) | | |
| 960 | | DEVICE_SPEC *def_ptr; | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/tmeas.c

DATE

5/23/89

PAGE #

TIME

6:14:50 pm

9/140

```

LINE # SOURCE TEXT
961 INSTANCE_INFO *instance;
962 u_long timeout;
963 u_char event_type;
964 u_short event_pin_number;
965 u_char event_pin_count;
966 FULL_VALUE *steady_state_result;
967 PTRN_BITS LONGWORD *ident_inconsistent_pins;
968 FULL_VALUE *temp_steady_state_result;
969 TM_PIN_INFO *tm_pin_info_table;
970 PTRN_BITS LONGWORD *ident_change;
971 u_char *changed_dac;
972 {
973
974
975 /*
976  * Measure delay of all output pins which change as a result of the pin
977  * event pin number(s). The type of the event (EVAL_EVENT or STORE_EVENT) is
978  * given in "event_type". If event_type == EVAL_EVENT then there could be
979  * multiple eval pins that change on this particular pattern seq. We need to
980  * add the delay of the eval pins to the measured pin delay that we found.
981  */
982 PTRN_BITS LONGWORD ident_all_pin_change[MAX_UNIT_COUNT];
983 PTRN_BITS LONGWORD *ident_all_pin_change_ptr;
984 PTRN_BITS LONGWORD *last_sample_data_ptr;
985 PTRN_BITS LONGWORD *last_sample_hiz_ptr;
986 PTRN_BITS LONGWORD *last_sample_unknown_ptr;
987 PTRN_BITS LONGWORD *steady_state_data_ptr;
988 PTRN_BITS LONGWORD *steady_state_hiz_ptr;
989 PTRN_BITS LONGWORD *steady_state_unknown_ptr;
990 PTRN_BITS LONGWORD *ident_ioa_ptr;
991 PTRN_BITS LONGWORD *ident_outputs_ptr;
992 PTRN_BITS LONGWORD *two_state_result[MAX_UNIT_COUNT];
993 DAB_INFO *dab_ptr;
994 TM_PIN_INFO *tm_pin_info;
995 EXTRA_DEVICE_SPEC *extra_def_ptr;
996 UMS_OFFSET *ums_ptr;
997 u_long ident_change_word;
998 u_long mask;
999 u_long min_event_pin_delay;
1000 u_long max_event_pin_delay;
1001 u_long min_delay;
1002 u_long max_delay;
1003 u_long save_min_delay;
1004 u_long save_max_delay;
1005 u_long z_to_1_transition;
1006 u_long measurement_error;
1007 short pins_to_be_measured_head;
1008 short all_pins_to_be_measured_head = -1;
1009 short pin_number;
1010 u_short min_event_pin_number;
1011 u_char resolution;
1012 u_char total_pins_to_be_measured = 0;
1013 u_char i;
1014 u_char total_unit;
1015 u_char unitno;
1016 u_char wordno;
1017 char hitno;
1018 u_char measurement_mode;
1019
1020 /* COMMENTED CODE
1021 char pin_name_list[MAX_PIN_NAME_LIST];
1022 u_char valid_pin_name_list = FALSE;
1023 END COMMENTED CODE
1024 */
1025 DPRINTF(("inside measure_delay\n"));
1026
1027 extra_def_ptr = (EXTRA_DEVICE_SPEC *) def_ptr->extra_data;
1028 dab_ptr = dab_list(instance->dab_info_index);
1029 total_unit = dab_ptr->unit_count;
1030
1031 #ifdef DBASE
1032 setup_final_tm_result(total_unit * 80);
1033 #endif
1034
1035 is_measure_delay = TRUE;
1036
1037 if (event_type == EVAL_EVENT) {
1038 /* Program the sample edge to start at data time */
1039 measurement_mode = extra_def_ptr->eval_event_mode;
1040 extra_def_ptr->edge_setting[6] = extra_def_ptr->edge_setting[8];
1041 if ((u_long) set_sample_trigger_mode(
1042 (u_long) extra_def_ptr->eval_event_mode) == FAILURE) {
1043 in_queue_message(ERROR_MSG, "Timing Generator error: set sample trigger mode");
1044 return (FAILURE);
1045 }
1046 } else {
1047 /* Program the sample edge to start at store time */
1048 measurement_mode = extra_def_ptr->store_event_mode;
1049 extra_def_ptr->edge_setting[6] = extra_def_ptr->edge_setting[7];
1050 if ((u_long) set_sample_trigger_mode(
1051 (u_long) extra_def_ptr->store_event_mode) == FAILURE) {
1052 in_queue_message(ERROR_MSG, "Timing Generator error: set sample trigger mode");
1053 return (FAILURE);
1054 }
1055 }
1056
1057 /* Sample the outputs at the max range of the finest resolution */
1058 if (set_edge_and_sample_setting(extra_def_ptr,
1059 (u_long) MAX_SAMPLE_RANGE - SWEEP_RANGE) == FAILURE)
1060 return (FAILURE);
1061
1062 if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE)
1063 return (FAILURE);
1064
1065 DPRINTF(("get end range result, resolution: %d\n", RESOLUTION_05_NS));
1066
1067 read_magic_full_sample_reg(instance, steady_state_result);
1068 read_magic_timing_sample_reg(instance, two_state_result,
1069 MAX_SAMPLE_RANGE - SWEEP_RANGE);
1070
1071
1072 ident_all_pin_change_ptr = ident_all_pin_change;
1073 last_sample_data_ptr = (PTRN_BITS LONGWORD *)
1074 instance->last_sample_value.data;
1075 last_sample_hiz_ptr = (PTRN_BITS LONGWORD *)
1076 instance->last_sample_value.hiz;
1077 last_sample_unknown_ptr = (PTRN_BITS LONGWORD *)
1078 instance->last_sample_value.unknown;
1079
1080

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/tmeas.c

DATE 5/23/89

PAGE #

TIME 6:14:50 pm

10/141

```

1081 steady_state_data_ptr = (PTM_BITS_LONGWORD *)
1082 steady_state_result_data;
1083 steady_state_hiz_ptr = (PTM_BITS_LONGWORD *)
1084 steady_state_result_hiz;
1085 steady_state_unknown_ptr = (PTM_BITS_LONGWORD *)
1086 steady_state_result_unknown;
1087
1088 ident_ios_ptr = (PTM_BITS_LONGWORD *) extra_def_ptr->ident_ios;
1089 ident_outputs_ptr = (PTM_BITS_LONGWORD *) extra_def_ptr->ident_outputs;
1090
1091 for (unitno = 0; unitno < total_unit; ++unitno) {
1092     for (wordno = 0; wordno < 3; ++wordno) {
1093         ident_all_pin_change_ptr->word[wordno] =
1094             last_sample_data_ptr->word[wordno];
1095         steady_state_data_ptr->word[wordno];
1096
1097         /* Don't measure transitions on I/O pins */
1098         ident_all_pin_change_ptr->word[wordno] |=
1099             (last_sample_hiz_ptr->word[wordno] |
1100              last_sample_unknown_ptr->word[wordno] |
1101              steady_state_hiz_ptr->word[wordno] |
1102              steady_state_unknown_ptr->word[wordno]);
1103
1104         /* Measure ZL to H transition on I/O pins */
1105         z_to_h_transition =
1106             ident_ios_ptr->word[wordno] &
1107             (last_sample_hiz_ptr->word[wordno] &
1108              last_sample_data_ptr->word[wordno]) &
1109             (~steady_state_hiz_ptr->word[wordno] |
1110              ~steady_state_unknown_ptr->word[wordno]);
1111         ident_all_pin_change_ptr->word[wordno] |= z_to_h_transition;
1112
1113         if (z_to_h_transition) {
1114             DPRINTF(("Found ZL to H transition\n"));
1115         }
1116         /* Measure HZ to L transition on I/O pins */
1117         h_to_l_transition =
1118             ident_ios_ptr->word[wordno] &
1119             (last_sample_hiz_ptr->word[wordno] &
1120              last_sample_data_ptr->word[wordno]) &
1121             (~steady_state_hiz_ptr->word[wordno] |
1122              ~steady_state_unknown_ptr->word[wordno]) &
1123             steady_state_data_ptr->word[wordno];
1124         ident_all_pin_change_ptr->word[wordno] |= h_to_l_transition;
1125
1126         if (h_to_l_transition) {
1127             DPRINTF(("Found HZ to L transition\n"));
1128         }
1129         /* Only measure timing on I/O pins and outputs pins */
1130         ident_all_pin_change_ptr->word[wordno] |=
1131             ident_ios_ptr->word[wordno] | ident_outputs_ptr->word[wordno];
1132
1133         ++ident_all_pin_change_ptr;
1134         ++last_sample_data_ptr;
1135         ++last_sample_hiz_ptr;
1136         ++last_sample_unknown_ptr;
1137         ++steady_state_data_ptr;
1138         ++steady_state_hiz_ptr;
1139         ++steady_state_unknown_ptr;
1140         ++ident_ios_ptr;
1141         ++ident_outputs_ptr;
1142     }
1143 }
1144
1145 /* Reset the bits in ident_all_pin_change corresponding to the
1146    event pin number. These pins will be forced added later. This is
1147    applicable only for IO pins because input pins have been masked off.
1148 */
1149 for (i = 0; i < event_pin_count; ++i) {
1150     pin_number = event_pin_number[i];
1151
1152     word_ptr = &pin_to_mask_offset(pin_number);
1153     unitno = word_ptr->unitno;
1154     wordno = word_ptr->wordno;
1155     bitno = word_ptr->bitno;
1156
1157     reset_ptrn_bit(((PTM_BITS *) ident_all_pin_change)[unitno],
1158                   wordno, (u_char) bitno);
1159 }
1160
1161 /* Find all pins which change values and link them in tm_pin_info_table */
1162 for (unitno = 0; unitno < total_unit; ++unitno) {
1163     for (wordno = 0; wordno < 3; ++wordno) {
1164         ident_change_word = ident_all_pin_change[unitno].word[wordno];
1165         for (bitno = 31; bitno >= 0; --bitno) {
1166             if (ident_change_word & (1 << bitno)) {
1167                 break;
1168             }
1169
1170             mask = bitno_to_mask(bitno);
1171
1172             if (ident_change_word & mask) {
1173                 ident_change_word ^= mask;
1174
1175                 --total_pins_to_be_measured;
1176
1177                 pin_number = CALC_PIN_NUMBER_LONG(unitno, wordno, bitno);
1178                 DPRINTF(("Link to tm_pin_info_table PN: %d\n", pin_number));
1179                 tm_pin_info = tm_pin_info_table[pin_number];
1180                 tm_pin_info->delay_found = FALSE;
1181                 tm_pin_info->next_pin_index = all_pins_to_be_measured_head;
1182                 all_pins_to_be_measured_head = pin_number;
1183             }
1184         }
1185     }
1186 }
1187
1188 /* Always measure the delay of the event pins */
1189 for (i = 0; i < event_pin_count; ++i) {
1190     event_pin = tm_pin_info_table[event_pin_number[i]];
1191 }
1192
1193
1194
1195
1196
1197
1198
1199
1200

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/tmeas.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:50 pm | 11/142 |

```

LINE # SOURCE TEXT
1201 event_pin->delay_found = FALSE;
1202 DPRINTF(("link eval pin to tm_pin_info_table PN: %d\n",
1203         event_pin_number[i]));
1204
1205 /* Use the minimum pin number to identify the event pin */
1206 if (event_pin_number[i] < min_event_pin_number)
1207     min_event_pin_number = event_pin_number[i];
1208
1209 event_pin->next_pin_index = all_pins_to_be_measured_head;
1210 all_pins_to_be_measured_head = event_pin_number[i];
1211
1212 }
1213 resolution = RESOLUTION_05_NS;
1214
1215 while (resolution != RESOLUTION_INVALID) {
1216
1217     /*
1218     * The actual pin delays are referenced to the event pin delay on the
1219     * same sample ramp. So we need to measure the event pins for each
1220     * resolution.
1221     */
1222
1223     /* Add the event pins to the list of pins to be measured */
1224     for (i = 0; i < event_pin_count; ++i) {
1225         event_pin = &tm_pin_info_table[event_pin_number[i]];
1226         event_pin->delay_found = FALSE;
1227         ++total_pins_to_be_measured;
1228     }
1229
1230 find_pins_to_be_measured(def_ptr,
1231                          tm_pin_info_table, two_state_result,
1232                          steady_state_result,
1233                          ident_all_pins_change,
1234                          event_pin_number,
1235                          event_pin_count,
1236                          spins_to_be_measured_head,
1237                          total_wait, resolution, measurement_mode);
1238
1239 pin_number = pins_to_be_measured_head;
1240 while (pin_number != -1) {
1241
1242     uwb_ptr = &pins_to_be_measured[pin_number];
1243     unitno = uwb_ptr->unitno;
1244     wordno = uwb_ptr->wordno;
1245     bitno = uwb_ptr->bitno;
1246
1247     tm_pin_info_table[pin_number].event_pin_number =
1248         min_event_pin_number;
1249
1250     if (read_ptrn_bit(&(PTN_BITS *) ident_inconsistent_pins)(unitno),
1251         wordno, (u_char) bitno) == 0) {
1252         if (measure_pin(instance, timeout,
1253                        tm_pin_info_table, (u_short) pin_number,
1254                        steady_state_result, ident_inconsistent_pins,
1255                        temp_steady_state_result, ident_change,
1256                        resolution, changed_dac, measurement_mode) == FAILURE)
1257             return (FAILURE);
1258     } else {
1259         DPRINTF(("PN: %d inconsistent --> not measured\n",
1260                 pin_number));
1261     }
1262
1263     --total_pins_to_be_measured;
1264     pin_number = tm_pin_info_table[pin_number].next_pin_to_be_measured;
1265 }
1266
1267 if (get_composite_eval_pin_delay(def_ptr, instance, tm_pin_info_table,
1268                                 event_pin_number, event_pin_count,
1269                                 measurement_mode, resolution,
1270                                 min_event_pin_delay, max_event_pin_delay) == FAILURE)
1271     return (FAILURE);
1272
1273 pin_number = pins_to_be_measured_head;
1274 while (pin_number != -1) {
1275     tm_pin_info = &tm_pin_info_table[pin_number];
1276
1277     /* Make believe that the delay is found, so that we will be
1278     * able to report delay not reached correctly.
1279     * We will look at the actual delay values to see whether
1280     * we can/can't measure the pin delay.
1281     */
1282     tm_pin_info->delay_found = TRUE;
1283
1284     lm_tag_get_sample_ramp_delay(resolution,
1285                                 tm_pin_info->min_threshold,
1286                                 min_delay,
1287                                 measurement_error);
1288
1289     min_delay -= measurement_error;
1290
1291     lm_tag_get_sample_ramp_delay(resolution,
1292                                 tm_pin_info->max_threshold,
1293                                 max_delay,
1294                                 measurement_error);
1295
1296     max_delay += measurement_error;
1297
1298     if ((min_delay < max_event_pin_delay) ||
1299         (max_delay < min_event_pin_delay)) {
1300         /*-----*/
1301         /* This is bogus delay (we failed to measure the pin transition).
1302         * This case happens when:
1303         * 1. We try to measure a transition from
1304         *    20 to 1 (or 21 to 0) on a DATA pin, but the simulator is
1305         *    sending a 1 (or 0) with the clock transition which causes
1306         *    the pin to switch from 20 to 1 (or 21 to 0). The transition
1307         *    from 20 to 1 (or 21 to 0) will already happen in the setup
1308         *    pattern, and therefore we will NOT be able to measure the
1309         *    transition.
1310         * 2. There is a misdeclaration of a STORE or EVAL pin
1311         *    in the Shell Software as shown below:
1312         *    DATA (a/b store/eval) _____
1313         *    STORE (causing event) _____
1314         *    OUT _____
1315         *    We fail to measure transition on OUT because it happens
1316         *    in the setup pattern.
1317         * 3. A pin is inconsistent between two sets of sample runs.
1318         *    Consider this scenario. Store pin A change to 1 and we
1319         *    sample output B to be 1, then store pin A change to 0
1320         *    and we sample output B to be 0. At this point we say
1321         *    that we should measure the transition on pin B.
1322

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/tmeas.c

DATE 5/23/89 PAGE #
TIME 6:14:50 pm 12/143

```

LINE # SOURCE TEXT
1321  * It's possible that we will fail to measure B because
1322  * this time pin B does not change to 1 on the rising
1323  * edge of pin A.
1324  .....
1325
1326  /* COERCED HERE.  case 3 above
1327  for (i = 0; i < event_pin_count; ++i) {
1328  if (pin_number == event_pin_number[i])
1329  break;
1330  }
1331
1332  if (i == event_pin_count) {
1333  pin_number = tm_pin_info->next_pin_to_be_measured;
1334  continue;
1335  }
1336
1337  ucb_ptr = tps to adjust offset(pin_number);
1338  ucb_ptr = ucb_ptr->unitno;
1339  ucb_ptr = ucb_ptr->unitno;
1340  hitno = ucb_ptr->hitno;
1341
1342  if (read_pmc_bits(PMC_BITS) != ident_inconsistent_pins){unitno;
1343  ucb_ptr = (u_char)hitno; if (i == 0) {
1344  pin_number = tm_pin_info->next_pin_to_be_measured;
1345  continue;
1346  }
1347
1348  if (read_pmc_bits(PMC_BITS) != ident_data_change(unitno;
1349  ucb_ptr, hitno) == 0) {
1350
1351  if (build_pin_name_list == FALSE) {
1352  build_pin_list(def_ptr, total_unit,
1353  (PMC_BITS_LONGWORD) &gbl_tm_ident_data_change,
1354  pin_name_list);
1355  build_pin_name_list = TRUE;
1356  }
1357
1358  in_name_change(PMC_BITS, "could not measure pin: is; one of these pins should be declared as store/eval: is",
1359  def_ptr->pin_table(pin_number).pin_name,
1360  pin_name_list);
1361  }
1362  END COERCED CODE */
1363
1364  } else {
1365
1366  save_min_delay = tm_pin_info->min_delay;
1367  save_max_delay = tm_pin_info->max_delay;
1368
1369  /* The real delay is referenced to the skew of the eval pins */
1370  tm_pin_info->min_delay = min_delay - max_event_pin_delay;
1371  tm_pin_info->max_delay = max_delay - min_event_pin_delay;
1372
1373  /* Adjust the delay further by the delay of the trace */
1374  adjust_measured_delay(def_ptr->pin_table, tm_pin_info, (u_short) pin_number, tm_pin_info_table);
1375
1376  if (tm_pin_info->delay_value_is_set == FALSE) {
1377  tm_pin_info->delay_value_is_set = TRUE;
1378  } else {
1379  /* Coalesce delays from several runs of timing measurements. */
1380  if (save_min_delay < tm_pin_info->min_delay)
1381  tm_pin_info->min_delay = save_min_delay;
1382  if (save_max_delay > tm_pin_info->max_delay)
1383  tm_pin_info->max_delay = save_max_delay;
1384  }
1385
1386  pin_number = tm_pin_info->next_pin_to_be_measured;
1387
1388  }
1389
1390  if (total_pins_to_be_measured == 0) {
1391  break;
1392  }
1393  /* Go to the next coarser resolution */
1394  switch (resolution) {
1395  case RESOLUTION_05_NS:
1396  resolution = RESOLUTION_10_NS;
1397  break;
1398  case RESOLUTION_10_NS:
1399  resolution = RESOLUTION_20_NS;
1400  break;
1401  case RESOLUTION_20_NS:
1402  resolution = RESOLUTION_40_NS;
1403  break;
1404  case RESOLUTION_40_NS:
1405  resolution = RESOLUTION_INVALID;
1406  continue;
1407  }
1408
1409  if (set_edge_and_sample_setting(extra_def_ptr,
1410  (u_long) resolution,
1411  (u_long) MAX_SAMPLE_RANGE - SWEEP_RANGE) == FAILURE)
1412  return (FAILURE);
1413
1414  DPRINTF(("get and range result: resolution: %d\n", resolution));
1415
1416  if (play_ptr_seq(instance, timeout, changed_dec) == FAILURE)
1417  return (FAILURE);
1418
1419  read_magic_full_sample_res(instance, temp_steady_state_result);
1420  read_magic_timing_sample_res(instance, two_state_result);
1421  MAX_SAMPLE_RANGE - SWEEP_RANGE);
1422
1423  /*
1424  * Note that "ident_change" below will be overwritten by get_result()
1425  * which we are going to call later on. This is OK since we only
1426  * collecting the "ident_inconsistent_pins. The above note does not apply
1427  * anymore since we are collecting the ident_change is get_result().
1428  */
1429
1430  check_consistency(instance,
1431  steady_state_result, temp_steady_state_result,
1432  ident_inconsistent_pins, ident_change,
1433  total_unit);
1434
1435  }
1436
1437  /* Set the delays for event pins to 0 */
1438  for (i = 0; i < event_pin_count; ++i) {
1439  event_pin = tm_pin_info_table[event_pin_number[i]];
1440  event_pin->delay_value_is_set = FALSE;
1441  event_pin->min_delay = 0;

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/tmeas.c

DATE 5/23/89
TIME 6:14:50 pm

PAGE #
13/144

```

1441 event_pin->max_delay = 0;
1442 }
1443
1444 /*
1445  * Check that all pins are measured. Note that "event_pin" is pointing to
1446  * the last event pin.
1447  */
1448 pin_number = event_pin->next_pin_index;
1449 while (pin_number != -1) {
1450     tm_pin_info = tm_pin_info_table(pin_number);
1451     if (tm_pin_info->delay_found == FALSE) {
1452         uwb_ptr = tps_to_ahart_offset(pin_number);
1453         unitno = uwb_ptr->unitno;
1454         wordno = uwb_ptr->wordno;
1455         bitno = uwb_ptr->bitno;
1456
1457         if (read_ptr_bit(&(amp_ptr->bits) ident_inconsistent_pins)(unitno),
1458             wordno, (u_char) bitno) == 0)
1459             in_queue_message(WARNING_MSG, "pin: %s of instance: %s delay not reached",
1460                             def_ptr->pin_table(pin_number).pin_name,
1461                             instance->device_info_string);
1462     }
1463     pin_number = tm_pin_info->next_pin_index;
1464 }
1465
1466 is_measure_delay = FALSE;
1467 DPRINTF(("Exiting measure_delay\n"));
1468 return (SUCCESS);
1469 }
1470
1471 find_pins_to_be_measured(def_ptr,
1472                          tm_pin_info_table, end_range_two_state_result,
1473                          steady_state_result, ident_all_pin_change,
1474                          event_pin_number, event_pin_count,
1475                          pins_to_be_measured_head,
1476                          total_unit, resolution, measurement_mode)
1477
1478 DEVICE_SPEC *def_ptr;
1479 TM_PIN_INFO tm_pin_info_table;
1480 PTRN_BITS_LONGWORD *steady_state_result;
1481 FULL_VALUE *ident_all_pin_change;
1482 PTRN_BITS_LONGWORD *event_pin_number;
1483 u_short *event_pin_count;
1484 u_char *pins_to_be_measured_head;
1485 short *total_unit;
1486 u_char *resolution;
1487 u_char *measurement_mode;
1488
1489 {
1490     PTRN_BITS_LONGWORD *steady_state_data_ptr;
1491     PTRN_BITS_LONGWORD *ident_change_ptr;
1492     PTRN_BITS_LONGWORD *ident_change[MAX_UNIT_COUNT];
1493     TM_PIN_INFO *tm_pin_info;
1494     UWB_OFFSET *uwb_ptr;
1495     u_long *ident_change_word;
1496     u_long *mask;
1497     u_short *pin_number;
1498     u_char *min_range;
1499     u_char *min_sample_ramp0;
1500     u_char *min_sample_ramp1;
1501     u_char *min_sample_ramp2;
1502     u_char *min_sample_ramp3;
1503     u_char *i;
1504     u_char *unitno;
1505     u_char *wordno;
1506     char *bitno;
1507
1508     DPRINTF(("inside find_pins_to_be_measured\n"));
1509
1510     in_tm_get_minimum_sample_ramp_setting(&min_sample_ramp0,
1511                                           &min_sample_ramp1,
1512                                           &min_sample_ramp2,
1513                                           &min_sample_ramp3);
1514
1515     switch (resolution) {
1516         case RESOLUTION_05_MS:
1517             min_range = min_sample_ramp0 + SWEET_RANGE;
1518             break;
1519         case RESOLUTION_10_MS:
1520             min_range = min_sample_ramp1 + SWEET_RANGE;
1521             break;
1522         case RESOLUTION_20_MS:
1523             min_range = min_sample_ramp2 + SWEET_RANGE;
1524             break;
1525         case RESOLUTION_40_MS:
1526             min_range = min_sample_ramp3 + SWEET_RANGE;
1527             break;
1528         default:
1529             min_range = min_sample_ramp3 + SWEET_RANGE;
1530             break;
1531     }
1532
1533     steady_state_data_ptr = (PTRN_BITS_LONGWORD *) steady_state_result->data;
1534     ident_change_ptr = (PTRN_BITS_LONGWORD *) ident_change;
1535
1536     /*
1537      * Identify the pins that have attained their steady state value at the end
1538      * of the sample ramp.
1539      */
1540     for (unitno = 0; unitno < total_unit; ++unitno) {
1541         for (wordno = 0; wordno < 3; ++wordno) {
1542             ident_change_ptr[word[wordno]] =
1543                 (steady_state_data_ptr[word[wordno]] -
1544                  end_range_two_state_result[word[wordno]]) &
1545                 ident_all_pin_change[word[wordno]];
1546         }
1547         ++steady_state_data_ptr;
1548         ++ident_change_ptr;
1549         ++end_range_two_state_result;
1550         ++ident_all_pin_change;
1551     }
1552
1553     steady_state_data_ptr = (PTRN_BITS_LONGWORD *) steady_state_result->data;
1554     pins_to_be_measured_head = -1;
1555     for (unitno = 0; unitno < total_unit; ++unitno) {
1556         for (wordno = 0; wordno < 3; ++wordno) {
1557             ident_change_word = ident_change[unitno].word[wordno];
1558             for (bitno = 31; bitno >= 0; --bitno) {
1559                 if (ident_change_word == 0)
1560                     break;
1561                 mask = bitno to mask[bitno];
1562             }
1563         }
1564     }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/tmeas.c

DATE 5/23/89 PAGE #
TIME 6:14:50 pm 14/145

```

1561
1562
1563     if (idest_change_word & mask) {
1564         idest_change_word = mask;
1565
1566         pin_number = CALC_PIN_NUMBER_LONG(unitno, wordno, bitno);
1567
1568         DPRINTF(("measure this pin in cur res PW: %d\n",
1569             pin_number));
1570
1571         tm_pin_info = tm_pin_info_table(pin_number);
1572
1573         if (tm_pin_info->delay_found == FALSE) {
1574             tm_pin_info->expected_value =
1575                 read_ptr_bit_long(&steady_state_data_ptr(unitno),
1576                     wordno, (u_char) bitno);
1577             tm_pin_info->min_threshold = min_range;
1578             tm_pin_info->max_threshold = MAX_SAMPLE_RANGE -
1579                 SWEEP_RANGE;
1580
1581             tm_pin_info->next_pin_to_be_measured =
1582                 "pin to be measured head";
1583             "pin to be measured head" = pin_number;
1584         } else {
1585             DPRINTF(("PW: %d delay is already found, not measured\n",
1586                 pin_number));
1587         }
1588     }
1589 }
1590
1591 /* Link the event pins to the next_pin_to_be_measured. */
1592 for (i = 0; i < event_pin_count; ++i) {
1593     pin_number = event_pin_number[i];
1594
1595     uwb_ptr = tpe_to_short_offset(pin_number);
1596     unitno = uwb_ptr->unitno;
1597     wordno = uwb_ptr->wordno;
1598     bitno = uwb_ptr->bitno;
1599
1600     DPRINTF(("measure this pin in cur res PW: %d\n",
1601         pin_number));
1602
1603     tm_pin_info = tm_pin_info_table(pin_number);
1604     tm_pin_info->expected_value =
1605         read_ptr_bit(&steady_state_result->data(unitno),
1606             wordno, (u_char) bitno);
1607
1608     if ((measurement_mode == EARLYSAMPLETRIGGERMODE) &&
1609         ((def_ptr->pin_table(pin_number).clk_format == R1) ||
1610         (def_ptr->pin_table(pin_number).clk_format == R0))) {
1611         tm_pin_info->min_threshold =
1612             ((EXTRA_DEVICE_SPEC *) def_ptr->extra_data->xl_or_x2_min_threshold;
1613     } else {
1614         tm_pin_info->min_threshold = min_range;
1615     }
1616
1617     tm_pin_info->max_threshold = MAX_SAMPLE_RANGE - SWEEP_RANGE;
1618
1619     tm_pin_info->next_pin_to_be_measured =
1620         "pin to be measured head";
1621     "pin to be measured head" = pin_number;
1622 }
1623
1624 DPRINTF(("exiting find_pins_to_be_measured\n"));
1625 }
1626
1627
1628
1629
1630
1631 NAME:
1632 round_trip_trace_delay -- Returns the round trip delay for this DAB.
1633
1634 PURPOSE:
1635 round_trip_trace_delay calculates the delay attributed to the propagation
1636 of the signal between the DUT and the magic chip. This includes the delay
1637 from the magic chip to the event pin, the delay from the result pin back to
1638 the magic chip and the PEL delay for driven pins.
1639
1640 INTERFACE:
1641
1642 u_short round_trip_trace_delay(pin_def, event_pin, result_pin)
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680

```

| PARAMETER | TYPE | DESCRIPTION |
|------------|-----------|---|
| pin_def | PIN_SPEC* | A pointer to the pin specification table. |
| event_pin | u_short | The pin number of the event pin. |
| result_pin | u_short | The pin number of the result pin. |

```

1675
1676
1677     return (PEL_DELAY
1678         + pin_def[event_pin].trace_delay
1679         + pin_def[result_pin].trace_delay);
1680

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/tmeas.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:50 pm | 15/146 |

```

1681 *
1682 * NAME:
1683 *   adjust_measured_delay -- Compute actual delay from measured delay.
1684 *
1685 * PURPOSE:
1686 *   The "adjust_measured_delay" adjusts the delay values for the
1687 *   maximum and minimum delays for "other" known delays not attributed
1688 *   to actual BVT. It also adjusts the delays attributed to the
1689 *   MAGIC IC delay effect.
1690 *   In the event that the measured delay is less than
1691 *   the "other" known delays, the measured delay will be set to zero.
1692 *
1693 * INTERFACE:
1694 *
1695 *   adjust_measured_delay(pin_def, tm_pin_info,
1696 *       result_pin, tm_pin_info_table)
1697 *
1698 *   PARAMETER      TYPE      DESCRIPTION
1699 *   -----
1700 *   pin_def         PIN_SPEC*  A pointer to the pin specification table.
1701 *   tm_pin_info     TM_PIN_INFO* A pointer to the timing measurement pin
1702 *       information table.
1703 *   result_pin      u_short     The pin number of the result pin.
1704 *   tm_pin_info_table TM_PIN_INFO* A pointer to the TM_PIN_INFO table
1705 *       to find the event pin's TM_PIN_INFO.
1706 *
1707 * CALLS:
1708 *   round_trip_trace_delay()
1709 *
1710 * EXTERNAL REFERENCES:
1711 *   None.
1712 *
1713 * RESTRICTIONS:
1714 *   No checking is done to validate any of the parameters.
1715 *
1716 * DESIGNER:
1717 *   Steve Parke
1718 *
1719 * MODIFICATIONS:
1720 *   05/09/89: RRM Added Magic delay adjustments
1721 *
1722 * .....
```

```

1723 *
1724 * adjust_measured_delay(pin_def, tm_pin_info, result_pin, tm_pin_info_table)
1725 * register PIN_SPEC *pin_def;
1726 * register TM_PIN_INFO *tm_pin_info;
1727 * u_short result_pin;
1728 * TM_PIN_INFO *tm_pin_info_table;
1729 * {
1730 * {
1731 *
1732 * /*
1733 * * The delay we measured is actually longer than it should be because of
1734 * * trace delays on the PCL and the DAB.
1735 * */
1736 * u_short event_pin = tm_pin_info->event_pin_number;
1737 * u_short trace_delay = round_trip_trace_delay(pin_def, event_pin, result_pin);
1738 * TM_PIN_INFO *event_tm_pin_info;
1739 * u_short magic_delay;
1740 *
1741 * DPRINTF(("Delay (%5s,%5s): ", pin_def[event_pin].pin_name, pin_def[result_pin].pin_name));
1742 * DPRINTF(("Before: %5d,%5d: ", tm_pin_info->min_delay, tm_pin_info->max_delay));
1743 *
1744 * tm_pin_info->min_delay = (trace_delay > tm_pin_info->min_delay)
1745 *     ? 0 : tm_pin_info->min_delay - trace_delay;
1746 * tm_pin_info->max_delay = (trace_delay > tm_pin_info->max_delay)
1747 *     ? 0 : tm_pin_info->max_delay - trace_delay;
1748 *
1749 * DPRINTF(("After: %5d,%5d: ", tm_pin_info->min_delay, tm_pin_info->max_delay));
1750 *
1751 * /* Adjust the delay further by the MAGIC IC effect. */
1752 * event_tm_pin_info = tm_pin_info_table[event_pin];
1753 * if (event_tm_pin_info->expected_value) {
1754 *     if (tm_pin_info->expected_value) {
1755 *         magic_delay = MAGIC_IIN_LOUT_DELAY;
1756 *     }
1757 *     else {
1758 *         magic_delay = MAGIC_IIN_OOUT_DELAY;
1759 *     }
1760 * }
1761 * else {
1762 *     if (tm_pin_info->expected_value) {
1763 *         magic_delay = MAGIC_OIN_LOUT_DELAY;
1764 *     }
1765 *     else {
1766 *         magic_delay = MAGIC_OIN_OOUT_DELAY;
1767 *     }
1768 * }
1769 *
1770 * tm_pin_info->min_delay =
1771 *     ((long)tm_pin_info->min_delay + (long)magic_delay < 0)
1772 *     ? 0 : tm_pin_info->min_delay + magic_delay;
1773 * tm_pin_info->max_delay =
1774 *     ((long)tm_pin_info->max_delay + (long)magic_delay < 0)
1775 *     ? 0 : tm_pin_info->max_delay + magic_delay;
1776 *
1777 * DPRINTF(("After2: %5d,%5d\n", tm_pin_info->min_delay, tm_pin_info->max_delay));
1778 * }
1779 *
1780 * measure_pin(instance, timeout, tm_pin_info_table, target_pin_number,
1781 *     steady_state_result, ident_inconsistent_pins,
1782 *     temp_steady_state_result, ident_change, resolution,
1783 *     changed_dac, measurement_mode)
1784 * {
1785 *     INSTANCE_INFO *instance;
1786 *     u_long timeout;
1787 *     TM_PIN_INFO *tm_pin_info_table;
1788 *     u_short target_pin_number;
1789 *     FULL_VALUE *steady_state_result;
1790 *     PTRN_BITS_LONGWORD *ident_inconsistent_pins;
1791 *     FULL_VALUE *temp_steady_state_result;
1792 *     PTRN_BITS_LONGWORD *ident_change;
1793 *     u_char resolution;
1794 *     u_char changed_dac;
1795 *     u_char measurement_mode;
1796 * {
1797 *     DEVICE_SPEC *dev_ptr;
1798 *     EXTRA_DEVICE_SPEC *extra_dev_ptr;
1799 *     DAB_INFO *dab_ptr;
1800 *     TM_PIN_INFO *tm_pin_info;
1801 *     target_pin;

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/tmeas.c | DATE 5/23/89 | PAGE # 16/147 |
|--|------------|--|-----------------|------------------|
| LINE # | | SOURCE TEXT | | |
| 1801 | UWB_OFFSET | *uwb_ptr, | | |
| 1802 | UWB_BITS | two_state_result[MAX_UNIT_COUNT], | | |
| 1803 | u_long | temp, | | |
| 1804 | short | pin_number, | | |
| 1805 | short | start, | | |
| 1806 | short | end, | | |
| 1807 | short | mid, | | |
| 1808 | u_char | total_unit, | | |
| 1809 | u_char | result, | | |
| 1810 | u_char | unitno, | | |
| 1811 | u_char | wordno, | | |
| 1812 | u_char | bitno, | | |
| 1813 | u_char | min_sample_ramp0, | | |
| 1814 | u_char | min_sample_ramp1, | | |
| 1815 | u_char | min_sample_ramp2, | | |
| 1816 | u_char | min_sample_ramp3, | | |
| 1817 | u_char | min_range, | | |
| 1818 | | | | |
| 1819 | | DPRINTF(("inside measure_pin PW: %d\n", target_pin_number)); | | |
| 1820 | | | | |
| 1821 | | dab_ptr = dab_list(instance->dab_info_index); | | |
| 1822 | | total_unit = dab_ptr->unit_count; | | |
| 1823 | | | | |
| 1824 | | def_ptr = instance->definition; | | |
| 1825 | | extra_def_ptr = (EXTRA_DEVICE_SPEC *) instance->definition->extra_data; | | |
| 1826 | | | | |
| 1827 | | target_pin = tm_pin_info_table[target_pin_number]; | | |
| 1828 | | | | |
| 1829 | | start = target_pin->min_threshold; | | |
| 1830 | | end = target_pin->max_threshold; | | |
| 1831 | | mid = start + (end - start) / 2; | | |
| 1832 | | | | |
| 1833 | | while (start <= end) { | | |
| 1834 | | /* Sample the pins at "mid" */ | | |
| 1835 | | | | |
| 1836 | | if (set_edge_and_sample_setting(extra_def_ptr, | | |
| 1837 | | (u_long) resolution, | | |
| 1838 | | (u_long) mid) == FAILURE) | | |
| 1839 | | return (FAILURE); | | |
| 1840 | | | | |
| 1841 | | DPRINTF(("getting two state sample; sample time: %d\n", mid)); | | |
| 1842 | | | | |
| 1843 | | if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) | | |
| 1844 | | return (FAILURE); | | |
| 1845 | | | | |
| 1846 | | #ifdef DEBUG | | |
| 1847 | | if (loop_till_key == TRUE) { | | |
| 1848 | | printf("looping for threshold: %d\n", mid); | | |
| 1849 | | debug_key = 0; | | |
| 1850 | | while (debug_key == 0) { | | |
| 1851 | | if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) | | |
| 1852 | | return (FAILURE); | | |
| 1853 | | | | |
| 1854 | | if (debug_key == (u_long) 't') { | | |
| 1855 | | loop_till_key = FALSE; | | |
| 1856 | | printf("stop loop\n"); | | |
| 1857 | | | | |
| 1858 | | debug_key = 0; | | |
| 1859 | | | | |
| 1860 | | #endif | | |
| 1861 | | | | |
| 1862 | | read_basic_full_sample_req(instance, temp_steady_state_result); | | |
| 1863 | | read_basic_timing_sample_req(instance, two_state_result, (u_short) mid); | | |
| 1864 | | | | |
| 1865 | | /* | | |
| 1866 | | * Note that "ident_change" below will be overwritten by get_result() | | |
| 1867 | | * which we are going to call later on. This is OK since we only | | |
| 1868 | | * collect the "ident_inconsistent_pins". The above note does not apply | | |
| 1869 | | * anymore since we are collecting the ident_change in get_result(). | | |
| 1870 | | */ | | |
| 1871 | | check_consistency(instance, | | |
| 1872 | | steady_state_result, temp_steady_state_result, | | |
| 1873 | | ident_inconsistent_pins, ident_change, | | |
| 1874 | | total_unit); | | |
| 1875 | | | | |
| 1876 | | | | |
| 1877 | | /* Narrow the range for all pins to be measured except the target pin */ | | |
| 1878 | | pin_number = target_pin->next_pin_to_be_measured; | | |
| 1879 | | while (pin_number != -1) { | | |
| 1880 | | | | |
| 1881 | | uwb_ptr = aps_to_short_offset(pin_number); | | |
| 1882 | | unitno = uwb_ptr->unitno; | | |
| 1883 | | wordno = uwb_ptr->wordno; | | |
| 1884 | | bitno = uwb_ptr->bitno; | | |
| 1885 | | | | |
| 1886 | | tm_pin_info = tm_pin_info_table[pin_number]; | | |
| 1887 | | | | |
| 1888 | | result = read_ptrn_bit(two_state_result[unitno], wordno, bitno); | | |
| 1889 | | | | |
| 1890 | | if (result == tm_pin_info->expected_value) { | | |
| 1891 | | if (mid < tm_pin_info->max_threshold) | | |
| 1892 | | tm_pin_info->max_threshold = mid; | | |
| 1893 | | } else { | | |
| 1894 | | if (mid > tm_pin_info->min_threshold) | | |
| 1895 | | tm_pin_info->min_threshold = mid; | | |
| 1896 | | } | | |
| 1897 | | | | |
| 1898 | | pin_number = tm_pin_info->next_pin_to_be_measured; | | |
| 1899 | | | | |
| 1900 | | | | |
| 1901 | | uwb_ptr = aps_to_short_offset(target_pin_number); | | |
| 1902 | | unitno = uwb_ptr->unitno; | | |
| 1903 | | wordno = uwb_ptr->wordno; | | |
| 1904 | | bitno = uwb_ptr->bitno; | | |
| 1905 | | | | |
| 1906 | | /* Adjust the range of the target pin */ | | |
| 1907 | | result = read_ptrn_bit(two_state_result[unitno], wordno, bitno); | | |
| 1908 | | if (result == target_pin->expected_value) { | | |
| 1909 | | target_pin->max_threshold = mid; | | |
| 1910 | | end = mid - 1; | | |
| 1911 | | } else { | | |
| 1912 | | target_pin->min_threshold = mid; | | |
| 1913 | | start = mid + 1; | | |
| 1914 | | } | | |
| 1915 | | | | |
| 1916 | | mid = start + (end - start) / 2; | | |
| 1917 | | | | |
| 1918 | | | | |
| 1919 | | if (target_pin->min_threshold > target_pin->max_threshold) { | | |
| 1920 | | /* Exchange the delay */ | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/tmeas.c

DATE 5/23/89 PAGE #
TIME 6:14:50 pm 17/148

```

1621 temp = target_pin->min_threshold;
1622 target_pin->min_threshold = target_pin->max_threshold;
1623 target_pin->max_threshold = temp;
1624 }
1625 DPRINTF(("delay found: %d - %d\n",
1626         target_pin->min_threshold, target_pin->max_threshold));
1627
1628 if ((measurement_mode == EARLYSAMPLETRIGGERMODE) &&
1629     ((def_ptr->pin_table[target_pin_number].clk_format == R1) ||
1630     (def_ptr->pin_table[target_pin_number].clk_format == R0))) {
1631     min_range = entire_def_ptr->ri_or_rz_min_threshold;
1632 } else {
1633     ln_two_get_minimum_sample_ramp_setting(&min_sample_ramp0,
1634                                             &min_sample_ramp1,
1635                                             &min_sample_ramp2,
1636                                             &min_sample_ramp3);
1637
1638     switch (resolution) {
1639     case RESOLUTION_05_NS:
1640         min_range = min_sample_ramp0;
1641         break;
1642     case RESOLUTION_10_NS:
1643         min_range = min_sample_ramp1;
1644         break;
1645     case RESOLUTION_20_NS:
1646         min_range = min_sample_ramp2;
1647         break;
1648     case RESOLUTION_40_NS:
1649         min_range = min_sample_ramp3;
1650         break;
1651     default:
1652         min_range = 0xfff;
1653         break;
1654     }
1655 }
1656
1657 uwb_ptr = apb_to_short_offset(target_pin_number);
1658 unitno = uwb_ptr->unitno;
1659 wordno = uwb_ptr->wordno;
1660 bitno = uwb_ptr->bitno;
1661
1662 result = read_ptr_bits(&two_state_result[unitno], wordno, bitno);
1663 if (result == target_pin->expected_value) {
1664     if (target_pin->min_threshold <= min_range + SWEEP_RANGE) {
1665         DPRINTF(("pin: %d --> delay not found\n", target_pin_number));
1666         target_pin->delay_found = FALSE;
1667     } else {
1668         target_pin->delay_found = TRUE;
1669     }
1670 } else {
1671     target_pin->delay_found = TRUE;
1672 }
1673
1674 DPRINTF(("delay found after sweep: %d - %d\n",
1675         target_pin->min_threshold, target_pin->max_threshold));
1676
1677 DPRINTF(("exiting measure_pin\n"));
1678 return (SUCCESS);
1679 }
1680
1681 #ifdef DBASE
1682 setup_final_tm_result(total_pin_count)
1683 u_short total_pin_count;
1684 {
1685     u_short i;
1686     long pin_number;
1687     u_long expected_time;
1688     char line[80];
1689
1690     DPRINTF(("enter final tm result, end with pin number --1\n"));
1691
1692     for (i = 0; i < total_pin_count; ++i) {
1693         tm_result_array[i].resolution = RESOLUTION_INVALID;
1694         tm_result_array[i].expected_time = MAX_SAMPLE_RAMP_RANGE - SWEEP_RANGE;
1695         tm_result_array[i].set_by_user = FALSE;
1696     }
1697
1698     while (1) {
1699         DPRINTF(("pin number: "));
1700         gets(line);
1701         sscanf(line, "%d", &pin_number);
1702
1703         if (pin_number == -1)
1704             break;
1705
1706         DPRINTF(("expected value: "));
1707         gets(line);
1708
1709         if (strcmp(line, "0") == 0) {
1710             tm_result_array[pin_number].steady_state_full_value = LOGIC_0;
1711             tm_result_array[pin_number].two_state_value = LOGIC_0;
1712         } else if (strcmp(line, "1") == 0) {
1713             tm_result_array[pin_number].steady_state_full_value = LOGIC_1;
1714             tm_result_array[pin_number].two_state_value = LOGIC_1;
1715         } else if (strcmp(line, "10") == 0) {
1716             tm_result_array[pin_number].steady_state_full_value = LOGIC_20;
1717             tm_result_array[pin_number].two_state_value = LOGIC_0;
1718         } else if (strcmp(line, "11") == 0) {
1719             tm_result_array[pin_number].steady_state_full_value = LOGIC_21;
1720             tm_result_array[pin_number].two_state_value = LOGIC_1;
1721         } else if (strcmp(line, "01") == 0) {
1722             tm_result_array[pin_number].steady_state_full_value = LOGIC_0;
1723             tm_result_array[pin_number].two_state_value = LOGIC_1;
1724         } else if (strcmp(line, "00") == 0) {
1725             tm_result_array[pin_number].steady_state_full_value = LOGIC_0;
1726             tm_result_array[pin_number].two_state_value = LOGIC_0;
1727         } else {
1728             DPRINTF(("illegal value. ignored\n"));
1729             continue;
1730         }
1731
1732         DPRINTF(("expected time: "));
1733         gets(line);
1734         sscanf(line, "%d", &expected_time);
1735
1736         tm_result_array[pin_number].resolution = RESOLUTION_05_NS;
1737         tm_result_array[pin_number].expected_time = expected_time;
1738         tm_result_array[pin_number].set_by_user = TRUE;
1739     }
1740 }
1741 #endif

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/tmeas.c | DATE 5/23/89 | PAGE # 18/149 |
|--|--|--|-----------------|------------------|
| LINE # | | SOURCE TEXT | | |
| 2041 | | accumulate_idest pins(source, dest, total_unit) | | |
| 2042 | | PTRN_BITS_LONGWORD *source; | | |
| 2043 | | PTRN_BITS_LONGWORD *dest; | | |
| 2044 | | u_char total_unit; | | |
| 2045 | | { | | |
| 2046 | | u_long *source_ptr; | | |
| 2047 | | u_long *dest_ptr; | | |
| 2048 | | u_char total_word; | | |
| 2049 | | u_char wordao; | | |
| 2050 | | { | | |
| 2051 | | total_word = total_unit * 3; | | |
| 2052 | | source_ptr = (u_long *) source; | | |
| 2053 | | dest_ptr = (u_long *) dest; | | |
| 2054 | | for (wordao = 0; wordao < total_word; ++wordao) | | |
| 2055 | | { | | |
| 2056 | | dest_ptr[wordao] = source_ptr[wordao]; | | |
| 2057 | | } | | |
| 2058 | | /* COMMENTED CODE | | |
| 2059 | | build_pin_list(def_ptr, total_unit, idest_data_change_ptr, pin_name_list) | | |
| 2060 | | DEVICE_SPEC *def_ptr; | | |
| 2061 | | u_char total_unit; | | |
| 2062 | | PTRN_BITS_LONGWORD *idest_data_change_ptr; | | |
| 2063 | | char *pin_name_list; | | |
| 2064 | | { | | |
| 2065 | | { | | |
| 2066 | | u_long mask; | | |
| 2067 | | u_long idest_data_change; | | |
| 2068 | | u_short unit_pin_number_offset; | | |
| 2069 | | u_short word_pin_number_offset; | | |
| 2070 | | u_short pin_number; | | |
| 2071 | | u_char unitao; | | |
| 2072 | | u_char wordao; | | |
| 2073 | | u_char bitao; | | |
| 2074 | | u_char max_string_length; | | |
| 2075 | | { | | |
| 2076 | | max_string_length = MAX_PIN_NAME_LIST - 5; | | |
| 2077 | | { | | |
| 2078 | | pin_name_list[0] = '\0'; | | |
| 2079 | | unit_pin_number_offset = 0; | | |
| 2080 | | for (unitao = 0; unitao < total_unit; ++unitao) { | | |
| 2081 | | { | | |
| 2082 | | word_pin_number_offset = unit_pin_number_offset + 79; | | |
| 2083 | | for (wordao = 0; wordao < 3; ++wordao) { | | |
| 2084 | | idest_data_change = idest_data_change_ptr[unitao].word[wordao]; | | |
| 2085 | | for (bitao = 31; bitao >= 0; --bitao) { | | |
| 2086 | | { | | |
| 2087 | | break; | | |
| 2088 | | mask = bitao < 31 ? mask (1 << bitao) : mask; | | |
| 2089 | | if (idest_data_change & mask) { | | |
| 2090 | | idest_data_change = mask; | | |
| 2091 | | pin_number = word_pin_number_offset + bitao - 31; | | |
| 2092 | | { | | |
| 2093 | | if (strlen(pin_name_list) + | | |
| 2094 | | strlen(def_ptr->pin_table[pin_number].pin_name) + 1 <= | | |
| 2095 | | max_string_length) { | | |
| 2096 | | (void)strcat(pin_name_list, | | |
| 2097 | | def_ptr->pin_table[pin_number].pin_name); | | |
| 2098 | | (void)strcat(pin_name_list, " "); | | |
| 2099 | | } | | |
| 2100 | | else { | | |
| 2101 | | (void)strcat(pin_name_list, "..."); | | |
| 2102 | | return(SUCCESS); | | |
| 2103 | | } | | |
| 2104 | | } | | |
| 2105 | | word_pin_number_offset += 32; | | |
| 2106 | | } | | |
| 2107 | | unit_pin_number_offset += 80; | | |
| 2108 | | } | | |
| 2109 | | return(SUCCESS); | | |
| 2110 | | } | | |
| 2111 | | } | | |
| 2112 | | END COMMENTED CODE | | |
| 2113 | | /* | | |
| 2114 | | get_composite_eval_pin_delay(def_ptr, instance, tm_pin_info_table, | | |
| 2115 | | event_pin_number, event_pin_count, | | |
| 2116 | | measurement_mode, resolution, | | |
| 2117 | | comp_eval_min_delay, comp_eval_max_delay) | | |
| 2118 | | { | | |
| 2119 | | DEVICE_SPEC *def_ptr; | | |
| 2120 | | INSTANCE_INFO *instance; | | |
| 2121 | | TM_PIN_INFO *tm_pin_info_table; | | |
| 2122 | | u_short event_pin_number; | | |
| 2123 | | u_char event_pin_count; | | |
| 2124 | | u_char measurement_mode; | | |
| 2125 | | u_char resolution; | | |
| 2126 | | long comp_eval_min_delay; | | |
| 2127 | | long comp_eval_max_delay; | | |
| 2128 | | { | | |
| 2129 | | { | | |
| 2130 | | u_char i; | | |
| 2131 | | { | | |
| 2132 | | TM_PIN_INFO *event_pin; | | |
| 2133 | | long min_event_pin_delay; | | |
| 2134 | | long max_event_pin_delay; | | |
| 2135 | | u_long measurement_error; | | |
| 2136 | | { | | |
| 2137 | | /* Convert the range found in measure_pin() to real delay in picosec */ | | |
| 2138 | | /* Find the min-max skew of all EVAL pins in terms of register setting */ | | |
| 2139 | | min_event_pin_delay = MAX_SAMPLE_RANGE - SKEW_RANGE; | | |
| 2140 | | max_event_pin_delay = 0; | | |
| 2141 | | { | | |
| 2142 | | if (measurement_mode == EDGE_TRIGGERMODE) { | | |
| 2143 | | { | | |
| 2144 | | for (i = 0; i < event_pin_count; ++i) { | | |
| 2145 | | { | | |
| 2146 | | event_pin = tm_pin_info_table[event_pin_number[i]]; | | |
| 2147 | | { | | |
| 2148 | | if (event_pin->delay_found == FALSE) { | | |
| 2149 | | in_queue_message(ERROR_MSG, "internal error: event pin: %s of instance: %s delay not found", | | |
| 2150 | | instance->device_info_string); | | |
| 2151 | | return (FAILURE); | | |
| 2152 | | } | | |
| 2153 | | if (event_pin->min_threshold < min_event_pin_delay) | | |
| 2154 | | min_event_pin_delay = event_pin->min_threshold; | | |
| 2155 | | if (event_pin->max_threshold > max_event_pin_delay) | | |
| 2156 | | max_event_pin_delay = event_pin->max_threshold; | | |
| 2157 | | } | | |
| 2158 | | } | | |
| 2159 | | { | | |
| 2160 | | /* Convert the min-max skew to picosec */ | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/tmeas.c

DATE 5/23/89
TIME 6:14:50 pm

PAGE #
19/150

```

LINE # SOURCE TEXT
2161  lm_tmq_get_sample_ramp_delay(resolution,
2162  (u_char) min_event_pin_delay,
2163  &min_event_pin_delay,
2164  &measurement_error);
2165  min_event_pin_delay += measurement_error;
2166
2167  lm_tmq_get_sample_ramp_delay(resolution,
2168  (u_char) max_event_pin_delay,
2169  &max_event_pin_delay,
2170  &measurement_error);
2171  max_event_pin_delay += measurement_error;
2172
2173  /* Using EARLY SAMPLE mode */
2174  for (i = 0; i < event_pin_count; ++i) {
2175      event_pin = &tm_pin_info_table[event_pin_number[i]];
2176
2177      if (resolution == RESOLUTION_05_NS) {
2178          if (event_pin->delay_found == FALSE) {
2179              lm_queue_message(ERROR_MSG, "internal error: event pin: %s of instance: %s delay not found",
2180              &event_pin->pin_name,
2181              instance->device_info_string);
2182              return (FAILURE);
2183          }
2184      } else {
2185          break;
2186      }
2187  }
2188
2189  if (i == event_pin_count) {
2190      /* All of the event pin delays are found. */
2191
2192      for (i = 0; i < event_pin_count; ++i) {
2193          event_pin = &tm_pin_info_table[event_pin_number[i]];
2194
2195          /* Save the delays in gbl_eval_min/max_delay[] */
2196          lm_tmq_get_sample_ramp_delay(resolution,
2197          event_pin->min_threshold,
2198          &gbl_eval_min_delay[i],
2199          &measurement_error);
2200          gbl_eval_min_delay[i] += measurement_error;
2201
2202          lm_tmq_get_sample_ramp_delay(resolution,
2203          event_pin->max_threshold,
2204          &gbl_eval_max_delay[i],
2205          &measurement_error);
2206          gbl_eval_max_delay[i] += measurement_error;
2207
2208          /* Calculate the worst case threshold */
2209          if (event_pin->min_threshold < min_event_pin_delay)
2210              min_event_pin_delay = event_pin->min_threshold;
2211          if (event_pin->max_threshold > max_event_pin_delay)
2212              max_event_pin_delay = event_pin->max_threshold;
2213      }
2214
2215      /* Convert the min-max skew to picoseconds */
2216      lm_tmq_get_sample_ramp_delay(resolution,
2217      (u_char) min_event_pin_delay,
2218      &min_event_pin_delay,
2219      &measurement_error);
2220      min_event_pin_delay += measurement_error;
2221
2222      lm_tmq_get_sample_ramp_delay(resolution,
2223      (u_char) max_event_pin_delay,
2224      &max_event_pin_delay,
2225      &measurement_error);
2226      max_event_pin_delay += measurement_error;
2227  } else {
2228      /*
2229       * Not all of the event pin delays are found. Use the delays from
2230       * previous resolution.
2231       */
2232
2233      min_event_pin_delay = MAXINT;
2234      max_event_pin_delay = 0;
2235
2236      for (i = 0; i < event_pin_count; ++i) {
2237          event_pin = &tm_pin_info_table[event_pin_number[i]];
2238
2239          if (gbl_eval_min_delay[i] < min_event_pin_delay)
2240              min_event_pin_delay = gbl_eval_min_delay[i];
2241          if (gbl_eval_max_delay[i] > max_event_pin_delay)
2242              max_event_pin_delay = gbl_eval_max_delay[i];
2243      }
2244  }
2245
2246  *comp_eval_min_delay = min_event_pin_delay;
2247  *comp_eval_max_delay = max_event_pin_delay;
2248
2249  return (SUCCESS);
2250
2251
2252

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

*

DATE

5/23/89

PAGE #

TIME

6:14:53 pm

1/151

```

LINE # SOURCE TEXT
1  /* SCOS_ID: util.c Rev 3.2, 5/9/89 at 11:02:18 */
2  #include "common.h"
3  #include "device.h"
4  #include "message.h"
5  #include "hardware.h"
6  #include "lm_rd_wr.h"
7  #include "mod_err.h"
8  #include "vvarm.h"
9  #include "septom.h"
10 #include "lmsrvr.h"
11 #include "function.h"
12 #include "protcmd.h"
13 #include "protans.h"
14 #include "tmg.h"
15 #include "lm_eff.h"
16
17 #ifdef MODELER
18 #include "vrtx.h"
19 #endif
20 #endif
21
22 #define lm_tmg_lane_select(ident_lane) (tmgptr->lane_enable = ident_lane)
23
24 extern LM_HARDWARE_ERROR modeler_error;
25 extern CONFIGURATION_ERRORS config_error;
26
27 extern TMG *tmgptr;
28
29 u_long lm_loop_patterns_count;
30
31 init()
32 {
33     PTRN_BITS *ptrn_bits_ptr;
34     u_short i, j;
35     u_long mask;
36     u_short pin_number;
37     u_char wait_pin_number;
38
39 #ifdef MODELER
40     /* ??? timer used to substitute VRTX library lm_timer() and lm_delay() */
41     lmittimer();
42 #endif
43
44     profile_init();
45
46     init_config_error();
47
48     /* initialize data structure */
49     for (i = 0; i < MAX_USER_COUNT; ++i) {
50         user_info_array[i] = (USER_INFO *)
51             DCALLOC((unsigned)sizeof(USER_INFO));
52         if (user_info_array[i] == NULL)
53             fatal_alloc_error();
54     }
55
56     /* initialize the dab_list array to NULL */
57     for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i)
58         dab_list[i] = NULL;
59
60     /* initialize misc variables */
61     system_config = NULL;
62
63     for (i = 0; i < MAX_FUNCTION; i++)
64         function_array[i] = process_no_suc;
65
66     function_array[(CREATE_DEF_CMD - 2) / 2] = process_create_def_cmd;
67     function_array[(CREATE_INSTANCE_CMD - 2) / 2] = process_create_instance_cmd;
68     function_array[(CREATE_FAULT_CMD - 2) / 2] = process_create_fault_cmd;
69
70     function_array[(RELEASE_DEF_CMD - 2) / 2] = process_release_def_cmd;
71     function_array[(RELEASE_INSTANCE_CMD - 2) / 2] = process_release_instance_cmd;
72     function_array[(RELEASE_FAULT_CMD - 2) / 2] = process_release_fault_cmd;
73
74     function_array[(EVAL_CMD - 2) / 2] = process_eval_cmd;
75
76     function_array[(SAVE_DEF_CMD - 2) / 2] = process_save_def_cmd;
77     function_array[(SAVE_PTRN_CMD - 2) / 2] = process_save_ptrn_cmd;
78     function_array[(SAVE_PTRN_COUNT_CMD - 2) / 2] = process_save_ptrn_count_cmd;
79     function_array[(RESTORE_INST_CMD - 2) / 2] = process_restore_inst_cmd;
80     function_array[(RESTORE_PTRN_CMD - 2) / 2] = process_restore_ptrn_cmd;
81     function_array[(PTRN_HIST_CMD - 2) / 2] = process_ptrn_hist_cmd;
82
83     function_array[(INQ_MODELER_CMD - 2) / 2] = process_inq_modeler_cmd;
84     function_array[(INQ_USER_LIST_CMD - 2) / 2] = process_inq_user_list_cmd;
85     function_array[(INQ_USER_CMD - 2) / 2] = process_inq_user_cmd;
86     function_array[(INQ_LANE_CMD - 2) / 2] = process_inq_lane_cmd;
87     function_array[(INQ_PAN_CMD - 2) / 2] = process_inq_pan_cmd;
88     function_array[(INQ_PEL_CMD - 2) / 2] = process_inq_pel_cmd;
89     function_array[(INQ_DEVICE_LIST_CMD - 2) / 2] = process_inq_device_list_cmd;
90     function_array[(INQ_DEVICE_NAME_CMD - 2) / 2] = process_inq_device_name_cmd;
91     function_array[(INQ_DEVICE_CMD - 2) / 2] = process_inq_device_cmd;
92     function_array[(INQ_DAB_CMD - 2) / 2] = process_inq_dab_cmd;
93     function_array[(INQ_DAB_LOC_CMD - 2) / 2] = process_inq_dab_loc_cmd;
94     function_array[(INQ_INSTANCE_CMD - 2) / 2] = process_inq_instance_cmd;
95     function_array[(INQ_FAULT_CMD - 2) / 2] = process_inq_fault_cmd;
96
97     function_array[(INQ_AVAIL_PTRN_CMD - 2) / 2] = process_inq_avail_ptrn_cmd;
98
99     function_array[(MEASUREMENT_CMD - 2) / 2] = process_measurement_cmd;
100
101     function_array[(LOOP_PTRN_CMD - 2) / 2] = process_loop_ptrn_cmd;
102
103     function_array[(RESET_INST_CMD - 2) / 2] = process_reset_inst_cmd;
104
105     function_array[(TEST_NETWORK_CMD - 2) / 2] = process_test_network_cmd;
106
107     function_array[(ABORT_CMD - 2) / 2] = process_abort_cmd;
108
109     function_array[(BEGIN_SESSION_CMD - 2) / 2] = process_begin_session_cmd;
110
111     function_array[(LABEL_DAB_CMD - 2) / 2] = process_label_dab_cmd;
112
113     function_array[(EVAL_CONTROL_CMD - 2) / 2] = process_eval_control_cmd;
114
115     function_array[(REBOOT_CMD - 2) / 2] = process_reboot_cmd;
116     function_array[(SHUTDOWN_CMD - 2) / 2] = process_shutdown_cmd;
117
118     function_array[(CHECK_DABDEF_CMD - 2) / 2] = process_check_dabdef_cmd;
119
120

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89
TIME 6:14:53 pm

PAGE #
2/152

```

LINE #          SOURCE TEXT
121  function_array((READ_CMD      - 2) / 2) = process_lm_read;
122  function_array((WRITE_CMD     - 2) / 2) = process_lm_write;
123
124  function_array((SAVE_DEF_CMD   - 2) / 2) = process_save_def_cmd;
125  function_array((PASSWORD_CMD  - 2) / 2) = process_password_cmd;
126
127  /* initialize feedback block count array */
128  for (i = 0; i < MAX_LANE_COUNT; ++i) {
129      for (j = 0; j < MAX_FAN_COUNT; ++j) {
130          fb_block_count_array[i][FAN_MAX_ADDR[j]] = MAXINT;
131          fb_block_count_array[i].feedback_block_count[j] = 0;
132      }
133  }
134
135  /* initialize the dummy pattern */
136  gbl_dummy_ptr = (PTRN_BITS *)DCALLOC((unsigned)MAX_UNIT_COUNT,
137                                       (unsigned)sizeof(PTRN_BITS));
138  if (gbl_dummy_ptr == NULL)
139      fatal_alloc_error();
140
141  ptrn_bits_ptr = gbl_dummy_ptr;
142  for (i = 0; i < MAX_UNIT_COUNT; ++i) {
143      set_pel_ctl(ptrn_bits_ptr->ctl, PEL_CTL_SELECT_PEL);
144      ++ptrn_bits_ptr;
145  }
146
147  /* initialize Word Mask Table which is used to convert bit number to
148  * bit mask. */
149  for (i = 0; mask = 1; i < 32; ++i, mask <= 1) {
150      bitso_to_mask[i] = mask;
151  }
152
153  /* initialize global full value structure */
154  allocate_full_value(&gbl_new_sample_value);
155  allocate_full_value(&gbl_steady_state_result);
156  allocate_full_value(&gbl_temp_steady_state_result);
157
158  /* initialize BIT TO LOGIC TABLE */
159  bit_to_logic_table[0x0] = LOGIC_0;
160  bit_to_logic_table[0x1] = LOGIC_1;
161  bit_to_logic_table[0x2] = LOGIC_30;
162  bit_to_logic_table[0x3] = LOGIC_31;
163
164  bit_to_logic_table[0x4] = LOGIC_20;
165  bit_to_logic_table[0x5] = LOGIC_21;
166  bit_to_logic_table[0x6] = LOGIC_20;
167  bit_to_logic_table[0x7] = LOGIC_21;
168
169  bit_to_logic_table[0x8] = LOGIC_U;
170  bit_to_logic_table[0x9] = LOGIC_U;
171  bit_to_logic_table[0xa] = LOGIC_U;
172  bit_to_logic_table[0xb] = LOGIC_U;
173  bit_to_logic_table[0xc] = LOGIC_U;
174  bit_to_logic_table[0xd] = LOGIC_U;
175  bit_to_logic_table[0xe] = LOGIC_U;
176  bit_to_logic_table[0xf] = LOGIC_U;
177
178  /* initialize pin number to unitno, wordno, bitno offset array */
179  for (pin_number = 0; pin_number < MAX_PIN_COUNT; ++pin_number) {
180      ps_to_short_offset[pin_number].unitno = pin_number / 80;
181      unit_pin_number = pin_number % 80;
182      ps_to_short_offset[pin_number].wordno = (79 - unit_pin_number) / 16;
183      ps_to_short_offset[pin_number].bitno = 15 - (79 - unit_pin_number) % 16;
184  }
185
186  #ifdef DBASE
187  /* initialize data base */
188  db_init();
189  #endif
190
191  allocate_full_value(full_value_ptr);
192  FULL_VALUE *full_value_ptr;
193  {
194      full_value_ptr->data =
195          (PTRN_BITS *)DCALLOC((unsigned)MAX_UNIT_COUNT,
196                               (unsigned)sizeof(PTRN_BITS));
197      if (full_value_ptr->data == NULL)
198          fatal_alloc_error();
199
200      full_value_ptr->hiz =
201          (PTRN_BITS *)DCALLOC((unsigned)MAX_UNIT_COUNT,
202                               (unsigned)sizeof(PTRN_BITS));
203      if (full_value_ptr->hiz == NULL)
204          fatal_alloc_error();
205
206      full_value_ptr->unknown =
207          (PTRN_BITS *)DCALLOC((unsigned)MAX_UNIT_COUNT,
208                               (unsigned)sizeof(PTRN_BITS));
209      if (full_value_ptr->unknown == NULL)
210          fatal_alloc_error();
211
212      full_value_ptr->soft =
213          (PTRN_BITS *)DCALLOC((unsigned)MAX_UNIT_COUNT,
214                               (unsigned)sizeof(PTRN_BITS));
215      if (full_value_ptr->soft == NULL)
216          fatal_alloc_error();
217  }
218
219
220
221  SYSTEM_INFO *
222  new_system_info()
223  {
224      SYSTEM_INFO *temp;
225      u_char i;
226
227      temp = (SYSTEM_INFO *)DCALLOC((unsigned)1,
228                                     (unsigned)sizeof(SYSTEM_INFO));
229      if (temp == NULL)
230          fatal_alloc_error();
231
232      for (i = 0; i < MAX_LANE_COUNT; ++i)
233          temp->lane[i] = NULL;
234      return(temp);
235  }
236
237  LANE_INFO *
238  new_lane_info()
239  {
240

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:53 pm | 3/153 |

```

LINE # SOURCE TEXT
241 LANE_INFO *temp;
242 u_char i;
243
244 temp = (LANE_INFO *)DCALLOC((unsigned)1, (unsigned)sizeof(LANE_INFO));
245 if (temp == NULL)
246     fatal_alloc_error();
247
248 temp->pac_present = FALSE;
249
250 for (i=0; i < MAX_PAM_COUNT; i++)
251     temp->pam[i] = NULL;
252 for (i=0; i < MAX_SLOT_COUNT; i++)
253     temp->pel[i] = NULL;
254 return(temp);
255
256
257 PEL_INFO *
258 new_pel_info()
259 {
260     PEL_INFO *temp;
261
262     temp = (PEL_INFO *)DCALLOC((unsigned)1, (unsigned)sizeof(PEL_INFO));
263     if (temp == NULL)
264         fatal_alloc_error();
265     return(temp);
266 }
267
268
269 DAB_INFO *
270 new_dab_info()
271 {
272     /* creates the DAB info */
273
274     DAB_INFO *temp;
275     u_char i;
276
277     temp = (DAB_INFO *)DCALLOC((unsigned)1, (unsigned)sizeof(DAB_INFO));
278     if (temp == NULL)
279         fatal_alloc_error();
280
281     temp->used_as_private = FALSE;
282
283     for (i = 0; i <= MAX_SEGMENT_PER_DEVICE; ++i)
284         temp->segment[i] = NULL;
285     return(temp);
286 }
287
288
289 rls_dab_info(dab_ptr)
290 DAB_INFO *dab_ptr;
291 {
292     u_char i;
293
294     for (i = 0; i <= MAX_SEGMENT_PER_DEVICE; ++i)
295         if (dab_ptr->segment[i] != NULL)
296             DFREE((char *)dab_ptr->segment[i]);
297     DFREE((char *)dab_ptr);
298 }
299
300
301 enter_dab_info(loc_dab_list, dab_ptr, laneso, slotso)
302 DAB_INFO *loc_dab_list[];
303 DAB_INFO *dab_ptr;
304 u_short laneso;
305 u_short slotso;
306 {
307     u_char index;
308
309     index = laneso * MAX_SLOT_COUNT + slotso;
310
311     DPRINTF(("inside enter_dab_info, name: %s laneso: %d slotso: %d index: %d\n",
312         dab_ptr->part_name, laneso, slotso, index));
313     loc_dab_list[index] = dab_ptr;
314 }
315
316
317
318
319
320 find_dab(loc_dab_list, name, device_type)
321 DAB_INFO *loc_dab_list[];
322 char *name;
323 u_char device_type; /* PRIVATE or PUBLIC */
324 {
325     /*
326      * Find the dab name in the device list.
327      * Return the index to dab_list if the name is found otherwise return -1
328      */
329     DAB_INFO *dab_ptr;
330     u_short i;
331
332     DPRINTF(("inside find_dab\n"));
333
334     for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i) {
335         dab_ptr = loc_dab_list[i];
336         if (dab_ptr != NULL)
337             if (strcmp(dab_ptr->part_name, name) == 0) {
338                 if (device_type == PRIVATE) {
339                     if ((dab_ptr->used_as_private == FALSE) &&
340                         ((dab_ptr->act_inst_count + dab_ptr->act_var_count) == 0))
341                     return(i);
342                 }
343                 else {
344                     if (dab_ptr->used_as_private == FALSE)
345                         return(i);
346                 }
347             }
348     }
349     return(-1);
350 }
351
352
353
354 PAM_INFO *
355 new_pam_info()
356 {
357     PAM_INFO *temp;
358
359     temp = (PAM_INFO *)DCALLOC((unsigned)1, (unsigned)sizeof(PAM_INFO));
360     if (temp == NULL)

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89
TIME 6:14:53 pm

PAGE #
4/154

```

LINE #          SOURCE TEXT
361      fatal_alloc_error();
362      return(temp);
363  }
364  }
365  SEGMENT_EL
366  new_segment()
367  {
368      SEGMENT_EL *temp;
369      temp = (SEGMENT_EL *)DCALLOC((unsigned)1, (unsigned)sizeof(SEGMENT_EL));
370      if (temp == NULL)
371          fatal_alloc_error();
372      return(temp);
373  }
374  }
375  }
376  }
377  }
378  set_last_in_lane(dab_ptr)
379  DAB_INFO *dab_ptr;
380  {
381      /* setup the following field in dab_info:
382       *   last_in_lane
383       *   lane_used field
384       *   lane_count
385       *   unit_count_per_lane
386       *   dummy_ptrn
387       *   ident_lane
388       */
389      char last_lane_found[MAX_LANE_COUNT];
390      char unit_count[MAX_LANE_COUNT];
391      u_char lanesno;
392      u_char i;
393      u_char ident_lane = 0;
394      u_char count;
395      u_char any_used_slotno_on_lane[MAX_LANE_COUNT];
396      for (lanesno = 0; lanesno < MAX_LANE_COUNT; ++lanesno) {
397          last_lane_found[lanesno] = 0;
398          unit_count[lanesno] = 0;
399      }
400      count = dab_ptr->unit_count;
401      for (i = 0; i < count; ++i) {
402          lanesno = dab_ptr->unit_location[i].lane_no;
403          dab_ptr->lane_used[lanesno] = 1;
404          if (last_lane_found[lanesno] == 0) {
405              dab_ptr->unit_location[i].last_in_lane = 1;
406              last_lane_found[lanesno] = 1;
407              ident_lane |= 1 << lanesno;
408          }
409          any_used_slotno_on_lane[lanesno] = dab_ptr->unit_location[i].slot_no;
410          ++unit_count[lanesno];
411      }
412      dab_ptr->ident_lane = ident_lane;
413      for (lanesno = 0; lanesno < MAX_LANE_COUNT; ++lanesno) {
414          if (dab_ptr->lane_used[lanesno]) {
415              ++dab_ptr->lane_count;
416              if (unit_count[lanesno] > dab_ptr->unit_count_per_lane)
417                  dab_ptr->unit_count_per_lane = unit_count[lanesno];
418              dab_ptr->dummy_ptrn[lanesno].word[2] =
419                  DUMMY_CTL_WORD(any_used_slotno_on_lane[lanesno]);
420          }
421      }
422  }
423  }
424  }
425  }
426  }
427  }
428  }
429  }
430  }
431  }
432  }
433  }
434  }
435  }
436  }
437  }
438  }
439  }
440  }
441  }
442  }
443  }
444  }
445  }
446  }
447  }
448  }
449  }
450  }
451  }
452  }
453  }
454  }
455  }
456  }
457  }
458  }
459  }
460  }
461  }
462  }
463  }
464  }
465  }
466  }
467  }
468  }
469  }
470  }
471  }
472  }
473  }
474  }
475  }
476  }
477  }
478  }
479  }
480  }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89

PAGE #

TIME 6:14:53 pm

5/155

```

LINE # SOURCE TEXT
481 inst_ptr = (INSTANCE_INFO *)temp->instance[i],
482 for (j = 0; j < (INST_TABLE_SIZE - 1); ++j) {
483     temp->instance[j] = (INSTANCE_INFO *)inst_ptr,
484     ++inst_ptr,
485 }
486 temp->instance[j] = NULL;
487 temp->free_inst_id = temp->instance;
488
489 /* create the definition table for this user */
490 temp->def_table_size = DEF_TABLE_SIZE;
491 temp->definition = (DEVICE_SPEC *)
492     EMALLOC((unsigned)(DEF_TABLE_SIZE * sizeof(DEVICE_SPEC *)));
493 if (temp->definition == NULL)
494     return(FAILURE);
495
496 def_ptr = (DEVICE_SPEC *)temp->definition[i],
497 for (j = 0; j < (DEF_TABLE_SIZE - 1); ++j) {
498     temp->definition[j] = (DEVICE_SPEC *)def_ptr,
499     ++def_ptr,
500 }
501
502 temp->definition[j] = NULL;
503 temp->free_def_id = temp->definition;
504
505 return(SUCCESS);
506 }
507
508 abort_user(user)
509 USER_INFO *user;
510 {
511     DAB_INFO *dab_ptr;
512     DEVICE_SPEC *definition;
513     INSTANCE_INFO *instance;
514     EXTRA_DEVICE_SPEC *extra_def_ptr;
515     u_long pattern_count;
516     u_long pattern_count1;
517     u_long longest_pattern_seq;
518     u_long cur_pattern_count;
519     u_short def_count;
520     u_short inst_count;
521     u_short fault_count;
522     u_short i;
523     u_char lane_count;
524     u_char lane;
525
526     DPRINTF(("inside abort_user\n"));
527
528     if (user->active == FALSE) {
529         lm_queue_message(ERROR_MSG, "internal error: attempt to release unused user");
530         return;
531     }
532
533     if (user->save_buffer != NULL) {
534         DPRINTF(user->save_buffer);
535     }
536
537     pattern_count1 = 0;
538     pattern_count = 0;
539     longest_pattern_seq = 0;
540     def_count = 0;
541     inst_count = 0;
542     fault_count = 0;
543     lane_count = 0;
544
545     /* Free instances */
546     for (i = 0; i < user->inst_table_size; ++i) {
547         instance = user->instance[i];
548
549         /* check if its a valid instance */
550         if (BOGUS_INSTANCE(user, instance))
551             continue;
552
553         extra_def_ptr = (EXTRA_DEVICE_SPEC *)instance->definition->extra_data;
554
555         for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {
556             if (extra_def_ptr->lane_used & 1 << lane)
557                 ++lane_count;
558         }
559
560         cur_pattern_count = instance->pattern_count * lane_count;
561
562         add_4((u_long)pattern_count, (u_long)0, cur_pattern_count);
563
564         if (cur_pattern_count > longest_pattern_seq)
565             longest_pattern_seq = cur_pattern_count;
566
567         return_all_ptr_block(instance);
568
569         if ((char)instance->dab_info_index != -1) {
570             dab_ptr = dab_list(instance->dab_info_index);
571
572             if (instance->is_fault == TRUE) {
573                 --dab_ptr->act_var_count;
574                 ++fault_count;
575             }
576             else {
577                 --dab_ptr->act_inst_count;
578                 ++inst_count;
579             }
580
581             if (dab_ptr->used_as_private == TRUE) {
582                 dab_ptr->used_as_private = FALSE;
583                 set_private_mode(dab_ptr, FALSE);
584             }
585
586             if ((dab_ptr->act_inst_count + dab_ptr->act_var_count) == 0)
587                 turn_off_in_use(dab_ptr);
588         }
589
590         xis_instance(user, i);
591     }
592
593     /* Free definitions */
594     for (i = 0; i < user->def_table_size; ++i) {
595         definition = user->definition[i];
596
597         /* check if its a valid definition */
598         if (BOGUS_DEFINITION(user, definition))
599             continue;
600

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:53 pm | 6/156 |

```

LINE # SOURCE TEXT
601
602 ++def_count,
603 rls_definition(user, i);
604 }
605
606 /* free the definition and instance tables */
607 DFREE((char *)user->definition);
608 DFREE((char *)user->instance);
609
610 user->active = FALSE;
611
612 write_user_stat(pattern_count, pattern_count1, longest_pattern_seq,
613 def_count, inst_count, fault_count);
614
615 DPRINTF(("exiting about user\n"));
616
617
618 new_and_link_definition(user, def_ptr, def_id_table_index)
619 USER_INFO *user;
620 DEVICE_SPEC *def_ptr;
621 u_short *def_id_table_index;
622 {
623
624     DEVICE_SPEC **slot_ptr;
625
626     DPRINTF(("inside new_and_link_definition\n"));
627
628     if (user->free_def_id == NULL)
629         if (extended_def_table(user) == FAILURE)
630             return(FAILURE);
631
632     slot_ptr = user->free_def_id;
633
634     *def_id_table_index = (u_short)
635         (((u_long)slot_ptr - (u_long)user->definition) / sizeof(DEVICE_SPEC));
636
637     user->free_def_id = (DEVICE_SPEC **)slot_ptr;
638     *slot_ptr = def_ptr;
639
640     def_ptr->extra_data = (char *)DCALLOC((unsigned)1,
641                                         (unsigned)sizeof(EXTRA_DEVICE_SPEC));
642     if (def_ptr->extra_data == NULL)
643         return(FAILURE);
644     ((EXTRA_DEVICE_SPEC *)def_ptr->extra_data)->dab_ok = TRUE;
645
646     return(SUCCESS);
647 }
648
649
650 allocate_initial_block(instance)
651 INSTANCE_INFO *instance;
652 {
653     /* Allocate a new block in each lane used by this instance.
654     * Setup the following fields:
655     *   LANE_ADDR_INFO.max_addr
656     *   LANE_ADDR_INFO.prev_max_addr
657     *   LANE_ADDR_INFO.last_unit_addr
658     *   LANE_ADDR_INFO.new_block_addr
659     *   INSTANCE_INFO.pattern_count
660     *   INSTANCE_INFO.unit_addr
661     */
662
663     DAB_INFO *dab_ptr;
664     LANE_ADDR_INFO *lane_ptr;
665     u_long ptrs_addr_to_assign(MAX_LANE_COUNT);
666     u_long temp_unit_addr;
667     u_char unit_processed(MAX_LANE_COUNT);
668     u_char unit_count(MAX_LANE_COUNT);
669     u_char lanes;
670     u_char unitno;
671     u_char i;
672     u_char dummy_unit_offset;
673     u_char junk;
674
675     DPRINTF(("inside allocate_initial_block\n"));
676
677     dab_ptr = dab_list(instance->dab_info_index);
678
679     for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {
680         lane_ptr = (LANE_ADDR_INFO *)instance->lane_addr[lane];
681         if (dab_ptr->lane_used[lane]) {
682             lane_ptr->prev_max_addr = lane_ptr->max_addr;
683
684             if (new_block(lane_ptr->max_addr,
685                         &junk,
686                         PATTERN_BLOCK,
687                         lane) == FAILURE) {
688                 lm_queue_message(ERROR_MSG, "out of Fast Pattern Memory");
689                 instance->failed_to_alloc_ptrs = TRUE;
690                 return(FAILURE);
691             }
692
693             lane_ptr->max_addr += BLOCK_ADDR_INC;
694
695             lane_ptr->last_unit_addr = lane_ptr->max_addr - BLOCK_ADDR_INC +
696                 (dab_ptr->unit_count_per_lane - 1) * PTRN_ADDR_INC;
697
698             lane_ptr->new_block_addr = 0
699
700             ptrs_addr_to_assign[lane] = lane_ptr->max_addr - BLOCK_ADDR_INC;
701         }
702         else {
703             ptrs_addr_to_assign[lane] = 0;
704         }
705     }
706
707     /* Initialize unit_count[] and unit_processed[] to be used later */
708     unit_count[lane] = 0;
709     unit_processed[lane] = 0;
710 }
711
712
713 /* Calculate the number of actual units in each lane */
714 for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno)
715     ++unit_count[dab_ptr->unit_location[unitno].lane_no];
716
717 /* Assign the addresses for the dummy patterns */
718 dummy_unit_offset = dab_ptr->unit_count + 1;
719 temp_unit_addr = instance->unit_addr[0][0];
720 instance->cur_unit_addr_index = 0;

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:53 pm | 7/157 |

```

LINE #          SOURCE TEXT
721  for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
722      if (dab_ptr->lane_used[lane0]) {
723          for (i = 0; i < dab_ptr->unit_count_per_lane -
724              unit_count[lane0]; ++i) {
725              temp_unit_addr[dummy_unit_offset++] =
726                  ptrs_addr_to_assign[lane0];
727              ptrs_addr_to_assign[lane0] += PTRN_ADDR_INC;
728          }
729      }
730      ptrs_addr_to_assign[lane0] += PTRN_ADDR_INC;
731  }
732  }
733  }
734  }
735  /* Assign the addresses for the actual units */
736  for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
737      temp_unit_addr[unitno] =
738          ptrs_addr_to_assign[dab_ptr->unit_location[unitno].lane_no];
739      ptrs_addr_to_assign[dab_ptr->unit_location[unitno].lane_no] +=
740          PTRN_ADDR_INC;
741  }
742  }
743  }
744  instance->pattern_count += dab_ptr->unit_count_per_lane;
745  return(SUCCESS);
746  }
747  }
748  }
749  }
750  load_dummy_patterns(instance)
751  INSTANCE_INFO *instance;
752  {
753      /* This routine builds some dummy patterns if necessary so that a proper
754       * branch can be made to the feedback sequence.
755       */
756      DAB_INFO *dab_ptr;
757      u_long temp_unit_addr;
758      u_short ptrs_count;
759      u_short dummy_ptrs_count;
760      i;
761      u_short quiet_patterns;
762      u_char total_unit;
763      u_char bits_per_pin = 1;
764      u_char unitno;
765      dab_ptr = dab_list(instance->dab_info_index);
766      quiet_patterns = 4 + 3 * (dab_ptr->unit_count_per_lane * bits_per_pin);
767      ptrs_count = quiet_patterns / dab_ptr->unit_count_per_lane;
768      if (quiet_patterns % dab_ptr->unit_count_per_lane != 0)
769          ++ptrs_count;
770      ptrs_count += (instance->definition->pre_seq_len +
771                  sizeof(PREAMBLE) / sizeof(PTRN_BITS)) *
772                  dab_ptr->unit_count_per_lane;
773      dummy_ptrs_count = 0;
774      if (ptrs_count % PTRN_PER_BLOCK != BRANCH_LATENCY) {
775          dummy_ptrs_count = (BRANCH_LATENCY - (ptrs_count % PTRN_PER_BLOCK) +
776                          dab_ptr->unit_count_per_lane) /
777                          dab_ptr->unit_count_per_lane;
778      }
779      if (((ptrs_count + dummy_ptrs_count) % dab_ptr->unit_count_per_lane) != 1) {
780          /* There will be an odd number of unit patterns before the feedback
781           * sequence. This means that the last unit pattern before the
782           * feedback sequence will be on an even pattern address. The branch
783           * command has to be specified 24 patterns before this last unit pattern,
784           * so it will also be on an even pattern address. Since the branch can only
785           * be made from an odd pattern address, therefore we need to add ONE
786           * UNIT of dummy patterns on each lane to make a proper branch.
787           */
788          adjust_patterns_addr(instance);
789          write_patterns_unit(instance, dab_ptr->dummy_ptrs);
790          /* Increment the address of each unit by PTRN_ADDR_INC. Also set the
791           * LANE_ADDR_INFO.last_unit_addr correctly because it was modified by
792           * adjust_patterns_addr(). We don't have to worry about going over
793           * current block max address because this is the first pattern in
794           * the block.
795           */
796          total_unit = dab_ptr->lane_count * dab_ptr->unit_count_per_lane;
797          temp_unit_addr = instance->unit_addr(instance->cur_unit_addr_index)[0];
798          for (unitno = 0; unitno < total_unit; ++unitno) {
799              temp_unit_addr[unitno] += PTRN_ADDR_INC;
800              if (dab_ptr->unit_location[unitno].last_in_lane)
801                  instance->lane_addr[dab_ptr->unit_location[unitno].lane_no].
802                      last_unit_addr = temp_unit_addr[unitno];
803          }
804          /* The pattern_count was subtracted in adjust_patterns_addr() */
805          instance->pattern_count += dab_ptr->unit_count_per_lane;
806      }
807      for (i = 0; i < dummy_ptrs_count; ++i) {
808          write_patterns(instance,
809                      instance->unit_addr(instance->cur_unit_addr_index)[0],
810                      &dummy_ptrs);
811          if (grow_patterns(instance) == FAILURE)
812              return(FAILURE);
813      }
814      return(SUCCESS);
815  }
816  }
817  }
818  }
819  }
820  }
821  }
822  }
823  }
824  }
825  }
826  }
827  }
828  }
829  }
830  }
831  load_dummy_patterns2(instance)
832  INSTANCE_INFO *instance;
833  {
834      /* This routine builds some dummy patterns so that the feedback signal
835       * is quiet before we branch to the feedback sequence.
836       */
837      DAB_INFO *dab_ptr;
838      u_short dummy_ptrs_count;
839      u_short i;

```


Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89

PAGE #

TIME 6:14:53 pm

8/158

```

LINE # SOURCE TEXT
841 u_short quiet_patterns;
842 u_char bits_per_pin = 1;
843
844 dab_ptr = dab_list(instance->dab_info_index);
845
846 quiet_patterns = 4 + 3 * (dab_ptr->unit_count_per_lane * bits_per_pin);
847
848 dummy_ptr_count = quiet_patterns / dab_ptr->unit_count_per_lane;
849 if (quiet_patterns % dab_ptr->unit_count_per_lane != 0)
850     ++dummy_ptr_count;
851
852 for (i = 0; i < dummy_ptr_count; ++i) {
853     if (grow_patterns(instance) == FAILURE)
854         return(FAILURE);
855
856     write_patterns(instance,
857                     instance->unit_addr(instance->cur_unit_addr_index)[0],
858                     &dummy_ptr_count);
859 }
860
861 return(SUCCESS);
862
863
864 allocate_feedback_block(instance)
865 INSTANCE_INFO *instance;
866 {
867     DAB_INFO *dab_ptr;
868     u_long addr_inc;
869     u_long temp_unit_addr;
870     u_char lane_no;
871     u_char unitno;
872
873     dab_ptr = dab_list(instance->dab_info_index);
874
875     /* Allocate feedback blocks on each lane used */
876     for (lane_no = 0; lane_no < MAX_LANE_COUNT; ++lane_no) {
877         if (dab_ptr->lane_used[lane_no]) {
878             if (new_block(instance->fb_block_addr[lane_no],
879                           instance->fb_block_size[lane_no],
880                           FEEDBACK_BLOCK,
881                           lane_no) == FAILURE) {
882                 instance->failed_to_alloc_ptr = TRUE;
883                 lm_queue_message(ERROR_MSG, "out of Fast Pattern Memory for feedback");
884                 return(FAILURE);
885             }
886         }
887     }
888
889     /* Calculate how much each unit address has to be incremented to get to
890     * a new block. This number is the same for each lane since up to this
891     * point the patterns grow at the same rate on each lane.
892     */
893     lane_no = dab_ptr->unit_location[0].lane_no;
894     addr_inc = instance->lane_addr[lane_no].max_addr -
895               (instance->lane_addr[lane_no].last_unit_addr -
896                (dab_ptr->unit_count_per_lane - 1) * PTRN_ADDR_INC);
897
898     for (lane_no = 0; lane_no < MAX_LANE_COUNT; ++lane_no) {
899         if (dab_ptr->lane_used[lane_no]) {
900             /* Set the branch command and enable feedback */
901             set_branch(instance->lane_addr[lane_no].last_unit_addr,
902                       instance->fb_block_addr[lane_no]);
903
904             /* Adjust PREV_MAX_ADDR and MAX_ADDR */
905             instance->lane_addr[lane_no].prev_max_addr =
906                 instance->lane_addr[lane_no].max_addr;
907
908             instance->lane_addr[lane_no].max_addr =
909                 instance->fb_block_addr[lane_no] +
910                 instance->fb_block_size[lane_no] * BLOCK_ADDR_INC;
911         }
912     }
913
914     temp_unit_addr = instance->unit_addr(instance->cur_unit_addr_index)[0];
915     for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
916         lane_no = dab_ptr->unit_location[unitno].lane_no;
917         temp_unit_addr[unitno] = instance->fb_block_addr[lane_no] +
918                                 temp_unit_addr[unitno] + addr_inc -
919                                 instance->lane_addr[lane_no].prev_max_addr;
920     }
921
922     return(SUCCESS);
923 }
924
925 set_ptrn_bit(ptrn_ptr, wordno, bitno)
926 PTRN_BITS *ptrn_ptr;
927 u_char wordno;
928 u_char bitno;
929 {
930     ptrn_ptr->word[wordno] |= (u_short)bitno_to_mask(bitno);
931 }
932
933 reset_ptrn_bit(ptrn_ptr, wordno, bitno)
934 PTRN_BITS *ptrn_ptr;
935 u_char wordno;
936 u_char bitno;
937 {
938     ptrn_ptr->word[wordno] &= (u_short)bitno_to_mask(bitno);
939 }
940
941 toggle_ptrn_bit(ptrn_ptr, wordno, bitno)
942 PTRN_BITS *ptrn_ptr;
943 u_char wordno;
944 u_char bitno;
945 {
946     ptrn_ptr->word[wordno] ^= (u_short)bitno_to_mask(bitno);
947 }
948
949 read_ptrn_bit(ptrn_ptr, wordno, bitno)
950 PTRN_BITS *ptrn_ptr;
951 u_char wordno;
952 u_char bitno;
953 {
954     ptrn_ptr->word[wordno] &= (u_short)bitno_to_mask(bitno);
955 }
956
957 read_ptrn_bit(ptrn_ptr, wordno, bitno)
958 PTRN_BITS *ptrn_ptr;
959 u_char wordno;
960 u_char bitno;

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/util.c | DATE 5/23/89 | PAGE # 9/159 |
|--|--|---|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 961 | | { return((ptrn_ptr->word[wordno] & (u_short)bitno_to_mask(bitno)) 0); | | |
| 962 | | } | | |
| 963 | | } | | |
| 964 | | set_pin_value(full_value_ptr, unitno, wordno, bitno, pin_value) | | |
| 965 | | FULL_VALUE *full_value_ptr; | | |
| 966 | | u_char unitno; | | |
| 967 | | u_char wordno; | | |
| 968 | | u_char bitno; | | |
| 969 | | u_char pin_value; | | |
| 970 | | { | | |
| 971 | | { | | |
| 972 | | switch (pin_value) { | | |
| 973 | | case LOGIC_0: | | |
| 974 | | reset_ptrn_bit(full_value_ptr->data [unitno], wordno, bitno); | | |
| 975 | | reset_ptrn_bit(full_value_ptr->hiz [unitno], wordno, bitno); | | |
| 976 | | reset_ptrn_bit(full_value_ptr->hiz [unitno], wordno, bitno); | | |
| 977 | | reset_ptrn_bit(full_value_ptr->unknown [unitno], wordno, bitno); | | |
| 978 | | reset_ptrn_bit(full_value_ptr->soft [unitno], wordno, bitno); | | |
| 979 | | break; | | |
| 980 | | case LOGIC_1: | | |
| 981 | | set_ptrn_bit (full_value_ptr->data [unitno], wordno, bitno); | | |
| 982 | | reset_ptrn_bit(full_value_ptr->hiz [unitno], wordno, bitno); | | |
| 983 | | reset_ptrn_bit(full_value_ptr->unknown [unitno], wordno, bitno); | | |
| 984 | | reset_ptrn_bit(full_value_ptr->soft [unitno], wordno, bitno); | | |
| 985 | | break; | | |
| 986 | | case LOGIC_20: | | |
| 987 | | reset_ptrn_bit(full_value_ptr->data [unitno], wordno, bitno); | | |
| 988 | | set_ptrn_bit (full_value_ptr->hiz [unitno], wordno, bitno); | | |
| 989 | | reset_ptrn_bit(full_value_ptr->unknown [unitno], wordno, bitno); | | |
| 990 | | reset_ptrn_bit(full_value_ptr->soft [unitno], wordno, bitno); | | |
| 991 | | break; | | |
| 992 | | case LOGIC_21: | | |
| 993 | | set_ptrn_bit (full_value_ptr->data [unitno], wordno, bitno); | | |
| 994 | | set_ptrn_bit (full_value_ptr->hiz [unitno], wordno, bitno); | | |
| 995 | | reset_ptrn_bit(full_value_ptr->unknown [unitno], wordno, bitno); | | |
| 996 | | reset_ptrn_bit(full_value_ptr->soft [unitno], wordno, bitno); | | |
| 997 | | break; | | |
| 998 | | case LOGIC_U: | | |
| 999 | | reset_ptrn_bit(full_value_ptr->data [unitno], wordno, bitno); | | |
| 1000 | | reset_ptrn_bit(full_value_ptr->hiz [unitno], wordno, bitno); | | |
| 1001 | | set_ptrn_bit (full_value_ptr->unknown [unitno], wordno, bitno); | | |
| 1002 | | reset_ptrn_bit(full_value_ptr->soft [unitno], wordno, bitno); | | |
| 1003 | | break; | | |
| 1004 | | case LOGIC_20: | | |
| 1005 | | reset_ptrn_bit(full_value_ptr->data [unitno], wordno, bitno); | | |
| 1006 | | reset_ptrn_bit(full_value_ptr->hiz [unitno], wordno, bitno); | | |
| 1007 | | reset_ptrn_bit(full_value_ptr->unknown [unitno], wordno, bitno); | | |
| 1008 | | set_ptrn_bit (full_value_ptr->soft [unitno], wordno, bitno); | | |
| 1009 | | break; | | |
| 1010 | | case LOGIC_21: | | |
| 1011 | | set_ptrn_bit (full_value_ptr->data [unitno], wordno, bitno); | | |
| 1012 | | reset_ptrn_bit(full_value_ptr->hiz [unitno], wordno, bitno); | | |
| 1013 | | reset_ptrn_bit(full_value_ptr->unknown [unitno], wordno, bitno); | | |
| 1014 | | set_ptrn_bit (full_value_ptr->soft [unitno], wordno, bitno); | | |
| 1015 | | break; | | |
| 1016 | | default: | | |
| 1017 | | break; | | |
| 1018 | | } | | |
| 1019 | | } | | |
| 1020 | | read_pin_value(full_value_ptr, unitno, wordno, bitno) | | |
| 1021 | | FULL_VALUE *full_value_ptr; | | |
| 1022 | | u_char unitno; | | |
| 1023 | | u_char wordno; | | |
| 1024 | | u_char bitno; | | |
| 1025 | | { | | |
| 1026 | | u_char index; | | |
| 1027 | | { | | |
| 1028 | | index = read_ptrn_bit(full_value_ptr-> | | |
| 1029 | | data[unitno], wordno, bitno) << 3 | | |
| 1030 | | read_ptrn_bit(full_value_ptr-> | | |
| 1031 | | hiz[unitno], wordno, bitno) << 2 | | |
| 1032 | | read_ptrn_bit(full_value_ptr-> | | |
| 1033 | | soft[unitno], wordno, bitno) << 1 | | |
| 1034 | | read_ptrn_bit(full_value_ptr-> | | |
| 1035 | | data[unitno], wordno, bitno); | | |
| 1036 | | return(bit_to_logic_table[index]); | | |
| 1037 | | } | | |
| 1038 | | } | | |
| 1039 | | } | | |
| 1040 | | /* | | |
| 1041 | | /* The following routines are cousins to the above routines. The difference | | |
| 1042 | | /* is that the wordno and bitno offsets are longword offsets. | | |
| 1043 | | */ | | |
| 1044 | | /* | | |
| 1045 | | set_ptrn_bit_long(ptrn_ptr, wordno, bitno) | | |
| 1046 | | PTRN_BITS_LONGWORD *ptrn_ptr; | | |
| 1047 | | u_char wordno; | | |
| 1048 | | u_char bitno; | | |
| 1049 | | { | | |
| 1050 | | { | | |
| 1051 | | ptrn_ptr->word[wordno] = bitno_to_mask(bitno); | | |
| 1052 | | } | | |
| 1053 | | } | | |
| 1054 | | reset_ptrn_bit_long(ptrn_ptr, wordno, bitno) | | |
| 1055 | | PTRN_BITS_LONGWORD *ptrn_ptr; | | |
| 1056 | | u_char wordno; | | |
| 1057 | | u_char bitno; | | |
| 1058 | | { | | |
| 1059 | | { | | |
| 1060 | | ptrn_ptr->word[wordno] ^= bitno_to_mask(bitno); | | |
| 1061 | | } | | |
| 1062 | | } | | |
| 1063 | | toggle_ptrn_bit_long(ptrn_ptr, wordno, bitno) | | |
| 1064 | | PTRN_BITS_LONGWORD *ptrn_ptr; | | |
| 1065 | | u_char wordno; | | |
| 1066 | | u_char bitno; | | |
| 1067 | | { | | |
| 1068 | | { | | |
| 1069 | | ptrn_ptr->word[wordno] ^= bitno_to_mask(bitno); | | |
| 1070 | | } | | |
| 1071 | | } | | |
| 1072 | | read_ptrn_bit_long(ptrn_ptr, wordno, bitno) | | |
| 1073 | | PTRN_BITS_LONGWORD *ptrn_ptr; | | |
| 1074 | | u_char wordno; | | |
| 1075 | | u_char bitno; | | |
| 1076 | | { | | |
| 1077 | | { | | |
| 1078 | | return((ptrn_ptr->word[wordno] & bitno_to_mask(bitno)) 0); | | |
| 1079 | | } | | |
| 1080 | | } | | |
| 1081 | | set_pin_value_long(full_value_ptr, unitno, wordno, bitno, pin_value) | | |
| 1082 | | FULL_VALUE *full_value_ptr; | | |
| 1083 | | u_char unitno; | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89 PAGE #
TIME 6:14:53 pm 10/160

```

1081 u_char wordao;
1082 u_char bitao;
1083 u_char pin_value;
1084 {
1085     switch (pin_value) {
1086     case LOGIC_0:
1087         reset_ptrn_bit_logg(&full_value_ptr->data [unitao], wordao, bitao);
1088         reset_ptrn_bit_logg(&full_value_ptr->hiz [unitao], wordao, bitao);
1089         reset_ptrn_bit_logg(&full_value_ptr->unknown [unitao], wordao, bitao);
1090         reset_ptrn_bit_logg(&full_value_ptr->soft [unitao], wordao, bitao);
1091         break;
1092     case LOGIC_1:
1093         set_ptrn_bit_logg (&full_value_ptr->data [unitao], wordao, bitao);
1094         reset_ptrn_bit_logg(&full_value_ptr->hiz [unitao], wordao, bitao);
1095         reset_ptrn_bit_logg(&full_value_ptr->unknown [unitao], wordao, bitao);
1096         reset_ptrn_bit_logg(&full_value_ptr->soft [unitao], wordao, bitao);
1097         break;
1098     case LOGIC_20:
1099         reset_ptrn_bit_logg(&full_value_ptr->data [unitao], wordao, bitao);
1100         set_ptrn_bit_logg (&full_value_ptr->hiz [unitao], wordao, bitao);
1101         reset_ptrn_bit_logg(&full_value_ptr->unknown [unitao], wordao, bitao);
1102         reset_ptrn_bit_logg(&full_value_ptr->soft [unitao], wordao, bitao);
1103         break;
1104     case LOGIC_21:
1105         set_ptrn_bit_logg (&full_value_ptr->data [unitao], wordao, bitao);
1106         set_ptrn_bit_logg (&full_value_ptr->hiz [unitao], wordao, bitao);
1107         reset_ptrn_bit_logg(&full_value_ptr->unknown [unitao], wordao, bitao);
1108         reset_ptrn_bit_logg(&full_value_ptr->soft [unitao], wordao, bitao);
1109         break;
1110     case LOGIC_U:
1111         reset_ptrn_bit_logg(&full_value_ptr->data [unitao], wordao, bitao);
1112         reset_ptrn_bit_logg(&full_value_ptr->hiz [unitao], wordao, bitao);
1113         set_ptrn_bit_logg (&full_value_ptr->unknown [unitao], wordao, bitao);
1114         reset_ptrn_bit_logg(&full_value_ptr->soft [unitao], wordao, bitao);
1115         break;
1116     case LOGIC_30:
1117         reset_ptrn_bit_logg(&full_value_ptr->data [unitao], wordao, bitao);
1118         reset_ptrn_bit_logg(&full_value_ptr->hiz [unitao], wordao, bitao);
1119         reset_ptrn_bit_logg(&full_value_ptr->unknown [unitao], wordao, bitao);
1120         set_ptrn_bit_logg (&full_value_ptr->soft [unitao], wordao, bitao);
1121         break;
1122     case LOGIC_31:
1123         set_ptrn_bit_logg (&full_value_ptr->data [unitao], wordao, bitao);
1124         reset_ptrn_bit_logg(&full_value_ptr->hiz [unitao], wordao, bitao);
1125         reset_ptrn_bit_logg(&full_value_ptr->unknown [unitao], wordao, bitao);
1126         set_ptrn_bit_logg (&full_value_ptr->soft [unitao], wordao, bitao);
1127         break;
1128     default:
1129         break;
1130     }
1131 }
1132
1133 read_pin_value_logg(full_value_ptr, unitao, wordao, bitao)
1134 FULL_VALUE *full_value_ptr;
1135 u_char unitao;
1136 u_char wordao;
1137 u_char bitao;
1138 {
1139     u_char index;
1140
1141     index = read_ptrn_bit_logg(&full_value_ptr->
1142         unknown[unitao], wordao, bitao) << 3 |
1143         read_ptrn_bit_logg(&full_value_ptr->
1144         hiz[unitao], wordao, bitao) << 2 |
1145         read_ptrn_bit_logg(&full_value_ptr->
1146         soft[unitao], wordao, bitao) << 1 |
1147         read_ptrn_bit_logg(&full_value_ptr->
1148         data[unitao], wordao, bitao);
1149
1150     return(bit_to_logic_table[index]);
1151 }
1152
1153 set_pel_ctl(word, value)
1154 u_short *word;
1155 u_char value;
1156 {
1157     *word = (*word & PEL_CTL_MASK) | (value << PEL_CTL_SHIFT);
1158 }
1159
1160 set_pel_user_bit(word)
1161 u_short *word;
1162 {
1163     *word |= PEL_USER_MASK;
1164 }
1165
1166 clear_pel_user_bit(word)
1167 u_short *word;
1168 {
1169     *word &= PEL_USER_MASK;
1170 }
1171
1172 set_pel_sel(word, value)
1173 u_short *word;
1174 u_char value;
1175 {
1176     *word = (*word & PEL_SEL_MASK) | (value << PEL_SEL_SHIFT);
1177 }
1178
1179 set_pel_br(word, value)
1180 u_short *word;
1181 u_char value;
1182 {
1183     *word = (*word & PEL_BRANCH_MASK) | (value << PEL_BRANCH_SHIFT);
1184 }
1185
1186 new_block(address, count, type, labao)
1187 u_long *address;
1188 u_char *count;
1189 u_char labao;
1190 u_char type;
1191 {
1192     /* if type == PATTERN_BLOCK then allocate 1 block on specified lane
1193     * if type == FEEDBACK_BLOCK then allocate n blocks on specified lane
1194     * Returns:
1195     * SUCCESS
1196     * the starting addr of the block, and count if can allocate block
1197     * FAILURE
1198     * if cannot allocate block
1199     */
1200 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89 PAGE #
TIME 6:14:53 pm 11/161

```

1201  * Note make sure when allocating feedback blocks, that the starting block
1202  * be on the x block boundary (where x is the number of blocks required
1203  * on the PAM type).
1204  */
1205
1206  u_long    temp;
1207  u_long    temp2;
1208  u_long    link_table_addr;
1209  u_long    value;
1210  u_long    value2;
1211  u_long    PAM_max_addr;
1212  u_short   block_no;
1213  u_char    i;
1214  u_char    panno;
1215  u_char    needed;
1216
1217  *address = NULL;
1218
1219  switch (type) {
1220  case PATTERN_BLOCK:
1221    if (free_block_list[laneno] == NULL)
1222      return(FAILURE);
1223
1224    temp = free_block_list[laneno];
1225
1226    DPRINTF(("New block addr: %8X\n", temp));
1227
1228    link_table_addr = ptol(temp);
1229
1230    DPRINTF(("link_table_addr: %8X\n", link_table_addr));
1231
1232    write_loc_long((u_long *)link_table_addr, (u_long)SEQ_END_BLOCK_FLAG);
1233
1234    free_block_list[laneno] = read_loc_long((u_long *)temp);
1235
1236    DPRINTF(("addr of prev link: %8X\n", free_block_list[laneno] + 4));
1237    write_loc_long((u_long *) (free_block_list[laneno] + 4), (u_long)NULL);
1238
1239    /*
1240    /*
1241    free_block_list[laneno] = temp->next;
1242    temp->prev = NULL;
1243
1244    *address = (u_long)temp;
1245    *count = 1;
1246
1247    break;
1248  case FEEDBACK_BLOCK:
1249    if (free_block_list[laneno] == NULL)
1250      return(FAILURE);
1251
1252    temp = free_block_list[laneno];
1253    while (temp != NULL) {
1254      for (panno = 0; panno < MAX_PAM_COUNT; ++panno) {
1255        if ((u_long)temp < fb_block_count_array[laneno].
1256            PAM_max_addr[panno])
1257          break;
1258      }
1259      needed = fb_block_count_array[laneno].
1260              feedback_block_count[panno];
1261      PAM_max_addr = fb_block_count_array[laneno].
1262                  PAM_max_addr[panno];
1263
1264      /* check if temp is in "needed" boundary */
1265      block_no = ptob(temp);
1266
1267      if (! (block_no & needed)) {
1268        /* Temp is pointing to a block which is on "needed" boundary.
1269        * Now check if ("needed" - 1) blocks after temp are free.
1270        */
1271        temp2 = temp + BLOCK_ADDR_INC;
1272        for (i = 0; i < (needed - 1); ++i) {
1273          /* break from the for loop if temp2 points to an address
1274          * outside the current PAM address range.
1275          */
1276          if ((u_long)temp2 >= PAM_max_addr)
1277            break;
1278
1279          link_table_addr = ptol(temp2);
1280
1281          value = read_loc_long((u_long *)link_table_addr) &
1282                BLOCK_NUMBER_MASK;
1283
1284          if (value != FREE_BLOCK_FLAG)
1285            break; /* break from the for loop */
1286          temp2 += BLOCK_ADDR_INC;
1287        }
1288        if (i == (needed - 1))
1289          break; /* break from the while loop */
1290      }
1291      temp = read_loc_long((u_long *)temp);
1292    }
1293
1294    if (temp == NULL)
1295      return(FAILURE);
1296
1297    /* We found usable feedback blocks */
1298    *address = (u_long)temp;
1299    *count = needed;
1300
1301    /* Take off "needed" blocks from the free list starting at temp. */
1302    for (i = 0; i < needed; ++i) {
1303      DPRINTF(("New FB block addr: %8X\n", temp));
1304
1305      link_table_addr = ptol(temp);
1306
1307      write_loc_long((u_long *)link_table_addr, (u_long)SEQ_END_BLOCK_FLAG);
1308      value = read_loc_long((u_long *) (temp + 4));
1309    }
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:53 pm | 12/162 |

```

1321     if (value == NULL) {
1322         free_block_list(laneno) = read_loc_long((u_long *)temp);
1323     }
1324     else {
1325         value = read_loc_long((u_long *)temp);
1326         value2 = read_loc_long((u_long *)temp + 4);
1327         write_loc_long((u_long *)value2, value);
1328     }
1329
1330     value = read_loc_long((u_long *)temp + 4);
1331     value2 = read_loc_long((u_long *)temp);
1332     write_loc_long((u_long *)value2 + 4, value);
1333
1334     temp = read_loc_long((u_long *)temp);
1335
1336     /*
1337     if (temp->prev == NULL)
1338         free_block_list(laneno) = temp->next;
1339     else
1340         temp->prev->next = temp->next;
1341
1342     temp->next->prev = temp->prev;
1343     temp = temp->next;
1344     */
1345 }
1346
1347 break;
1348 default:
1349     return(FAILURE);
1350 }
1351
1352 return(SUCCESS);
1353 }
1354
1355 release_block(address, laneno)
1356 u_long address;
1357 u_char laneno;
1358 {
1359     /* This routine releases one pattern block back to the correct free list.
1360     * "address" can point anywhere in the block.
1361     */
1362
1363     u_long link_table_addr;
1364     u_long block_addr;
1365     u_long temp;
1366     u_long temp2;
1367
1368     if (address == NULL)
1369         return;
1370
1371     address -= BLOCK_START_MASK;
1372
1373     DPRINTF(("Release block addr: %08X\n", address));
1374
1375     link_table_addr = ptol(address);
1376
1377     /* Mark this block free. */
1378     write_loc_long((u_long *)link_table_addr, (u_long)FREE_BLOCK_FLAG);
1379
1380     temp = link_table_addr - LINK_TABLE_ADDR_INC;
1381     temp2 = read_loc_long((u_long *)temp) & BLOCK_NUMBER_MASK;
1382     while (temp2 != FREE_BLOCK_FLAG) {
1383
1384         temp = LINK_TABLE_ADDR_INC;
1385         temp2 = read_loc_long((u_long *)temp) & BLOCK_NUMBER_MASK;
1386     }
1387
1388     if (temp == ((address & LANE_ADDR_MASK) + LANE_LINK_TABLE_OFFSET)) {
1389         /* TEMP is pointing to the link table location corresponding to block
1390         * number 0. So none of the blocks before this block is free.
1391         * Insert it at the head of the free list.
1392         */
1393
1394         if (free_block_list[laneno] != NULL) {
1395             write_loc_long((u_long *)free_block_list[laneno] + 4, address);
1396         }
1397         write_loc_long((u_long *)address, free_block_list[laneno]);
1398         write_loc_long((u_long *)address + 4, (u_long)NULL);
1399         free_block_list[laneno] = address;
1400     }
1401     else {
1402         /* Insert the block after temp */
1403
1404         block_addr = (temp & LANE_ADDR_MASK) +
1405             (temp << (BLOCK_NUMBER_SHIFT - 2));
1406
1407         DPRINTF(("insert block after block addr: %08X\n", block_addr));
1408
1409         temp2 = read_loc_long((u_long *)block_addr);
1410
1411         write_loc_long((u_long *)address, temp2);
1412         write_loc_long((u_long *)address + 4, block_addr);
1413
1414         write_loc_long((u_long *)block_addr, address);
1415         write_loc_long((u_long *)temp2 + 4, address);
1416     }
1417 }
1418
1419 read_branch_table_content(address)
1420 u_long address;
1421 {
1422     /* Read the content of the branch table corresponding to the block with
1423     * the given address.
1424     */
1425
1426     u_long link_table_addr;
1427     u_long value;
1428
1429     address -= BLOCK_START_MASK;
1430
1431     link_table_addr = ptol(address);
1432
1433     value = read_loc_long((u_long *)link_table_addr) & BLOCK_NUMBER_MASK;
1434
1435     return(value);
1436 }
1437
1438
1439
1440

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/util.c | DATE 5/23/89 | PAGE # 13/163 |
|--|--|--|-----------------|------------------|
| LINE # | | SOURCE TEXT | | |
| 1441 | | | | |
| 1442 | | | | |
| 1443 | | write_patterns(instance, unit_addr, ptrs_ptr) | | |
| 1444 | | INSTANCE_INFO *instance; | | |
| 1445 | | u_long unit_addr; | | |
| 1446 | | PTRN_BITS_LANE *ptrs_ptr; | | |
| 1447 | | { | | |
| 1448 | | /* Write the patterns pointed to by "ptrs_ptr" to address specified in | | |
| 1449 | | "unit_addr" array. | | |
| 1450 | | */ | | |
| 1451 | | DAB_INFO *dab_ptr; | | |
| 1452 | | u_long addr; | | |
| 1453 | | u_char unitno; | | |
| 1454 | | u_char laneso; | | |
| 1455 | | u_char total_unit; | | |
| 1456 | | | | |
| 1457 | | dab_ptr = dab_list(instance->dab_info_index); | | |
| 1458 | | total_unit = dab_ptr->unit_count; | | |
| 1459 | | | | |
| 1460 | | #ifdef DEBUG | | |
| 1461 | | if (unit_addr[0] == instance->lcycdb_addr[0]) { | | |
| 1462 | | DPRINTF(("LCYCDB :")); | | |
| 1463 | | } | | |
| 1464 | | else if (unit_addr[0] == instance->lcycmdb_addr[0]) { | | |
| 1465 | | DPRINTF(("LCYCMB :")); | | |
| 1466 | | } | | |
| 1467 | | else { | | |
| 1468 | | DPRINTF(("PTRN 408X", unit_addr[0])); | | |
| 1469 | | } | | |
| 1470 | | | | |
| 1471 | | for (unitno = 0; unitno < total_unit; ++unitno) { | | |
| 1472 | | DPRINTF((" 408X 408X 408X", ptrs_ptr[unitno].word[0], | | |
| 1473 | | ptrs_ptr[unitno].word[1], | | |
| 1474 | | ptrs_ptr[unitno].word[2]); | | |
| 1475 | | } | | |
| 1476 | | #endif | | |
| 1477 | | | | |
| 1478 | | for (unitno = 0; unitno < total_unit; ++unitno) { | | |
| 1479 | | addr = unit_addr[unitno]; | | |
| 1480 | | write_loc_long((u_long *)addr, | | |
| 1481 | | ptrs_ptr[unitno].word[0]); | | |
| 1482 | | write_loc_long((u_long *)(&addr + LANE_SEGMENT_B_OFFSET), | | |
| 1483 | | ptrs_ptr[unitno].word[1]); | | |
| 1484 | | write_loc_long((u_long *)(&addr + LANE_SEGMENT_C_OFFSET), | | |
| 1485 | | ptrs_ptr[unitno].word[2]); | | |
| 1486 | | } | | |
| 1487 | | | | |
| 1488 | | total_unit = dab_ptr->lane_count + dab_ptr->unit_count_per_lane; | | |
| 1489 | | for (unitno = 0; unitno < total_unit; ++unitno) { | | |
| 1490 | | addr = unit_addr[unitno]; | | |
| 1491 | | laneso = dab_ptr->unit_location[unitno].lane_no; | | |
| 1492 | | | | |
| 1493 | | write_loc_long((u_long *)addr, dab_ptr->dummy_ptr[laneso].word[0]); | | |
| 1494 | | write_loc_long((u_long *)(&addr + LANE_SEGMENT_B_OFFSET), | | |
| 1495 | | dab_ptr->dummy_ptr[laneso].word[1]); | | |
| 1496 | | write_loc_long((u_long *)(&addr + LANE_SEGMENT_C_OFFSET), | | |
| 1497 | | dab_ptr->dummy_ptr[laneso].word[2]); | | |
| 1498 | | } | | |
| 1499 | | | | |
| 1500 | | | | |
| 1501 | | | | |
| 1502 | | grow_patterns(instance) | | |
| 1503 | | INSTANCE_INFO *instance; | | |
| 1504 | | { | | |
| 1505 | | DAB_INFO *dab_ptr; | | |
| 1506 | | LANE_ADDR_INFO *lane_info; | | |
| 1507 | | u_long cur_unit_addr; | | |
| 1508 | | u_long new_unit_addr; | | |
| 1509 | | addr_inc_per_lane; | | |
| 1510 | | u_char laneso; | | |
| 1511 | | u_char unitno; | | |
| 1512 | | u_char total_unit; | | |
| 1513 | | u_char junk; | | |
| 1514 | | | | |
| 1515 | | dab_ptr = dab_list(instance->dab_info_index); | | |
| 1516 | | | | |
| 1517 | | instance->pattern_count += dab_ptr->unit_count_per_lane; | | |
| 1518 | | | | |
| 1519 | | addr_inc_per_lane = dab_ptr->unit_count_per_lane * PTRN_ADDR_INC; | | |
| 1520 | | | | |
| 1521 | | total_unit = dab_ptr->lane_count + dab_ptr->unit_count_per_lane; | | |
| 1522 | | | | |
| 1523 | | cur_unit_addr = instance->unit_addr[instance->cur_unit_addr_index][0]; | | |
| 1524 | | instance->cur_unit_addr_index = instance->cur_unit_addr_index + 1; | | |
| 1525 | | new_unit_addr = instance->unit_addr[instance->cur_unit_addr_index][0]; | | |
| 1526 | | for (unitno = 0; unitno < total_unit; ++unitno) { | | |
| 1527 | | lane_info = instance->lane_addr[dab_ptr->unit_location[unitno].lane_no]; | | |
| 1528 | | if ((cur_unit_addr[unitno] + addr_inc_per_lane) > lane_info->max_addr) { | | |
| 1529 | | if (lane_info->new_block_addr == 0) { | | |
| 1530 | | if (new_block(&lane_info->new_block_addr, | | |
| 1531 | | &junk, | | |
| 1532 | | PTRN_BLOCK, | | |
| 1533 | | dab_ptr->unit_location[unitno].lane_no) == FAILURE) { | | |
| 1534 | | lane_info->new_block_addr = 0; return(FAILURE); | | |
| 1535 | | lane_info->failed_to_alloc_ptr = TRUE; | | |
| 1536 | | return(FAILURE); | | |
| 1537 | | } | | |
| 1538 | | | | |
| 1539 | | set_branch(&lane_info->max_addr - PTRN_ADDR_INC, | | |
| 1540 | | &lane_info->new_block_addr); | | |
| 1541 | | | | |
| 1542 | | | | |
| 1543 | | new_unit_addr[unitno] = lane_info->new_block_addr + | | |
| 1544 | | cur_unit_addr[unitno] + addr_inc_per_lane - | | |
| 1545 | | lane_info->max_addr; | | |
| 1546 | | | | |
| 1547 | | | | |
| 1548 | | } | | |
| 1549 | | else { | | |
| 1550 | | new_unit_addr[unitno] = cur_unit_addr[unitno] + addr_inc_per_lane; | | |
| 1551 | | } | | |
| 1552 | | | | |
| 1553 | | if (dab_ptr->unit_location[unitno].last_in_lane) | | |
| 1554 | | lane_info->last_unit_addr = new_unit_addr[unitno]; | | |
| 1555 | | } | | |
| 1556 | | | | |
| 1557 | | for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) { | | |
| 1558 | | lane_info = instance->lane_addr[laneso]; | | |
| 1559 | | | | |
| 1560 | | if (lane_info->new_block_addr != 0) { | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89 PAGE #
TIME 6:14:53 pm 14/164

```

1561 lane_info->prev_max_addr = lane_info->max_addr;
1562 lane_info->new_addr = lane_info->new_block_addr + BLOCK_ADDR_INC;
1563 lane_info->new_block_addr = 0;
1564 }
1565 }
1566 }
1567 return(SUCCESS);
1568 }
1569
1570 set_branch(cur_ptrn_addr, current_block_addr)
1571 u_long cur_ptrn_addr;
1572 u_long current_block_addr;
1573 {
1574     /* Set the branch command BRANCH_LATENCY ptrns before the current pattern.
1575     * Modify the branch table appropriately.
1576     */
1577     u_long temp;
1578     u_long branch_addr;
1579     branch_addr = cur_ptrn_addr - BRANCH_LATENCY * PTRN_ADDR_INC +
1580                 LANE_SEGMENT_C_OFFSET;
1581     temp = read_loc_long((u_long *)branch_addr);
1582     temp = (temp & PEL_BRANCH_MASK) | (BRANCH_ALWAYS << PEL_BRANCH_SHIFT);
1583     write_loc_long((u_long *)branch_addr, temp);
1584     write_branch_table(cur_ptrn_addr, current_block_addr);
1585 }
1586
1587 unset_branch(cur_ptrn_addr)
1588 u_long cur_ptrn_addr;
1589 {
1590     /* Unset the branch command BRANCH_LATENCY ptrns before the current pattern.
1591     * Modify the branch table appropriately.
1592     */
1593     u_long temp;
1594     u_long branch_addr;
1595     branch_addr = cur_ptrn_addr - BRANCH_LATENCY * PTRN_ADDR_INC +
1596                 LANE_SEGMENT_C_OFFSET;
1597     temp = read_loc_long((u_long *)branch_addr);
1598     temp = (temp & PEL_BRANCH_MASK) | (BRANCH_NEVER << PEL_BRANCH_SHIFT);
1599     write_loc_long((u_long *)branch_addr, temp);
1600     /* Write 0 to the link table entry corresponding to cur_ptrn_addr
1601     * to indicate that this is the last block of the sequence.
1602     */
1603     write_branch_table(cur_ptrn_addr, (u_long)0);
1604 }
1605
1606 write_branch_table(source, destination)
1607 u_long source;
1608 u_long destination;
1609 {
1610     /* This routine writes the "destination" address into the branch table
1611     * location corresponding to the pattern block containing address
1612     * "source". I.e., "source" can point to be anywhere in the block.
1613     * Note that the contents of the link table is the "block number" of
1614     * the next block to jump to (not the actual address of the block).
1615     */
1616     u_long link_table_addr;
1617     source &= BLOCK_START_MASK;
1618     link_table_addr = ptol(source);
1619     write_loc_long((u_long *)link_table_addr,
1620                 (destination >> BLOCK_NUMBER_SHIFT) & BLOCK_NUMBER_MASK);
1621 }
1622
1623 grow_pattern_unit(instance)
1624 INSTANCE_INFO *instance;
1625 {
1626     /* This routine grows the last_in_lane unit pattern address on each lane
1627     * by just one unit.
1628     * It is assumed that there are enough rooms in the block for the pattern
1629     * to grow; it will not allocate new pattern block.
1630     */
1631     DAB_INFO *dab_ptr;
1632     LANE_ADDR_INFO *lane_info;
1633     u_char lanes;
1634     dab_ptr = dab_list(instance->dab_info_index);
1635     instance->pattern_count += 1;
1636     for (lanes = 0; lanes < MAX_LANE_COUNT; ++lanes) {
1637         lane_info = &instance->lane_addr[lanes];
1638         if (dab_ptr->lane_used[lanes]) {
1639             if ((lane_info->last_unit_addr + PTRN_ADDR_INC >=
1640                 lane_info->max_addr) {
1641                 lm_queue_message(ERROR_MSG, "Internal error: grow_pattern_unit pattern address overflow");
1642                 return(FAILURE);
1643             }
1644             lane_info->last_unit_addr += PTRN_ADDR_INC;
1645         }
1646     }
1647     return(SUCCESS);
1648 }
1649
1650 adjust_pattern_addr(instance)
1651 INSTANCE_INFO *instance;
1652 {
1653     /* This routine adjust the LANE_ADDR_INFO.last_unit_addr to point to
1654     * the address just past the last full lane pattern address.
1655     * It is assumed that subtracting the address does not make the address
1656     * go outside the block address boundary.
1657     */
1658     DAB_INFO *dab_ptr;

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/util.c | DATE 5/23/89 | PAGE # 15/165 |
|--|--|--|-----------------|------------------|
| LINE # | | SOURCE TEXT | | |
| 1681 | | u_char laneso; | | |
| 1682 | | dab_ptr = dab_list(instance->dab_info_index); | | |
| 1683 | | for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) { | | |
| 1684 | | if (dab_ptr->lane_used[laneso]) { | | |
| 1685 | | instance->lane_addr[laneso].last_unit_addr -- | | |
| 1686 | | (dab_ptr->unit_count_per_lane - 1) * PTRN_ADDR_INC; | | |
| 1687 | | } | | |
| 1688 | | instance->pattern_count = instance->pattern_count - | | |
| 1689 | | dab_ptr->unit_count_per_lane + 1; | | |
| 1690 | | } | | |
| 1691 | | write_patterns_unit(instance, ptrn_ptr) | | |
| 1692 | | INSTANCE_INFO *instance; | | |
| 1693 | | PTRN_BITS_LONGWORD *ptrn_ptr; | | |
| 1694 | | { | | |
| 1695 | | /* Use the addresses for the last in lane unit to write the 1 unit worth | | |
| 1696 | | * of patterns pointed by ptrn_ptr. | | |
| 1697 | | */ | | |
| 1698 | | DAB_INFO *dab_ptr; | | |
| 1699 | | LANE_ADDR_INFO *lane_info; | | |
| 1700 | | u_long addr; | | |
| 1701 | | u_char laneso; | | |
| 1702 | | dab_ptr = dab_list(instance->dab_info_index); | | |
| 1703 | | for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) { | | |
| 1704 | | lane_info = instance->lane_addr[laneso]; | | |
| 1705 | | if (dab_ptr->lane_used[laneso]) { | | |
| 1706 | | addr = lane_info->last_unit_addr; | | |
| 1707 | | write_loc_long((u_long *)addr, | | |
| 1708 | | ptrn_ptr[laneso].word(0)); | | |
| 1709 | | write_loc_long((u_long *) (addr + LANE_SEGMENT_B_OFFSET), | | |
| 1710 | | ptrn_ptr[laneso].word(1)); | | |
| 1711 | | write_loc_long((u_long *) (addr + LANE_SEGMENT_C_OFFSET), | | |
| 1712 | | ptrn_ptr[laneso].word(2)); | | |
| 1713 | | } | | |
| 1714 | | #ifdef DEBUG | | |
| 1715 | | DPRINTF(("PTRN UNIT 108X: 108X 108X 108X\n", addr, | | |
| 1716 | | ptrn_ptr[laneso].word(0), | | |
| 1717 | | ptrn_ptr[laneso].word(1), | | |
| 1718 | | ptrn_ptr[laneso].word(2)); | | |
| 1719 | | #endif | | |
| 1720 | | } | | |
| 1721 | | } | | |
| 1722 | | copy_patterns(source_addr, dest_addr, count) | | |
| 1723 | | u_long source_addr; | | |
| 1724 | | u_long dest_addr; | | |
| 1725 | | u_short count; | | |
| 1726 | | { | | |
| 1727 | | /* This routine copy "count" number of unit patterns from "source_addr" | | |
| 1728 | | * to "dest_addr". | | |
| 1729 | | */ | | |
| 1730 | | u_long cur_source_addr; | | |
| 1731 | | u_long cur_dest_addr; | | |
| 1732 | | u_long value; | | |
| 1733 | | u_short i; | | |
| 1734 | | { | | |
| 1735 | | DPRINTF(("copy_patterns: source_addr: 108X, dest_addr: 108X\n", | | |
| 1736 | | source_addr, dest_addr)); | | |
| 1737 | | cur_source_addr = source_addr; | | |
| 1738 | | cur_dest_addr = dest_addr; | | |
| 1739 | | for (i = 0; i < count; ++i) { | | |
| 1740 | | value = read_loc_long((u_long *)cur_source_addr); | | |
| 1741 | | write_loc_long((u_long *)cur_dest_addr, value); | | |
| 1742 | | value = | | |
| 1743 | | read_loc_long((u_long *) (cur_source_addr + LANE_SEGMENT_B_OFFSET)); | | |
| 1744 | | write_loc_long((u_long *) (cur_dest_addr + LANE_SEGMENT_B_OFFSET), value); | | |
| 1745 | | value = | | |
| 1746 | | read_loc_long((u_long *) (cur_source_addr + LANE_SEGMENT_C_OFFSET)); | | |
| 1747 | | write_loc_long((u_long *) (cur_dest_addr + LANE_SEGMENT_C_OFFSET), value); | | |
| 1748 | | cur_source_addr += PTRN_ADDR_INC; | | |
| 1749 | | cur_dest_addr += PTRN_ADDR_INC; | | |
| 1750 | | } | | |
| 1751 | | } | | |
| 1752 | | extend_inst_table(user) | | |
| 1753 | | USER_INFO *user; | | |
| 1754 | | { | | |
| 1755 | | INSTANCE_INFO **temp_ptr; | | |
| 1756 | | char *ptr; | | |
| 1757 | | long new_size; | | |
| 1758 | | u_long i; | | |
| 1759 | | { | | |
| 1760 | | new_size = user->inst_table_size + INST_TABLE_INCR; | | |
| 1761 | | ptr = (char *) | | |
| 1762 | | DREALLOC((char *)user->instance, | | |
| 1763 | | (unsigned)(new_size * sizeof(INSTANCE_INFO))); | | |
| 1764 | | if (ptr == NULL) | | |
| 1765 | | return(FAILURE); | | |
| 1766 | | user->instance = (INSTANCE_INFO **)ptr; | | |
| 1767 | | user->free_inst_id = (INSTANCE_INFO *) | | |
| 1768 | | user->instance[user->inst_table_size]; | | |
| 1769 | | temp_ptr = user->instance[user->inst_table_size + 1]; | | |
| 1770 | | for (i = user->inst_table_size; i < (new_size - 1); i++) { | | |
| 1771 | | { | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89

PAGE #

TIME 6:14:53 pm

16/166

```

1801 user->instance[i] = (INSTANCE_INFO *)temp_ptr;
1802 temp_ptr++;
1803 }
1804
1805 user->instance[i] = NULL;
1806 user->inst_table_size = new_size;
1807
1808 return(SUCCESS);
1809 }
1810
1811 extend_def_table(user)
1812 USER_INFO *user;
1813 {
1814     DEVICE_SPEC **temp_ptr;
1815     char *ptr;
1816     long new_size;
1817     u_long i;
1818
1819     new_size = user->def_table_size + DEF_TABLE_INCR;
1820
1821     ptr = (char *)
1822     DREALLOC((char *)user->definition,
1823             (unsigned)(new_size * sizeof(DEVICE_SPEC *)));
1824     if (ptr == NULL)
1825         return(FAILURE);
1826
1827     user->definition = (DEVICE_SPEC **)ptr;
1828
1829     user->free_def_id = user->definition[user->def_table_size];
1830
1831     temp_ptr = (DEVICE_SPEC **)
1832     user->definition[user->def_table_size + 1];
1833
1834     for (i = user->def_table_size; i < (new_size - 1); i++) {
1835         user->definition[i] = (DEVICE_SPEC *)temp_ptr;
1836         ++temp_ptr;
1837     }
1838
1839     user->definition[i] = NULL;
1840     user->def_table_size = new_size;
1841
1842     return(SUCCESS);
1843 }
1844
1845 new_and_link_instance(user, def_ptr, name_ptr,
1846                      unit_count, inst_id_table_index,
1847                      dab_info_index)
1848 USER_INFO *user;
1849 DEVICE_SPEC *def_ptr;
1850 char *name_ptr;
1851 u_short unit_count;
1852 u_short inst_id_table_index;
1853 char *dab_info_index;
1854 {
1855     INSTANCE_INFO *temp_ptr;
1856     INSTANCE_INFO *alot_ptr;
1857     PTRN_BITS *ptrn_bits_ptr;
1858     PIN_INFO *pin;
1859     DAB_INFO *dab_ptr;
1860     EXTRA_DEVICE_SPEC *extra_def_ptr;
1861     PTRN_BITS_LONGWORD *unit_ptr;
1862     PIN_SPEC *pin_spec_ptr;
1863     long pinno;
1864     u_char unitno;
1865     u_char wordno;
1866     u_char alloc_error = FALSE;
1867
1868     temp_ptr = (INSTANCE_INFO *)DCALLOC((unsigned)1,
1869                                         (unsigned)sizeof(INSTANCE_INFO));
1870     if (temp_ptr == NULL)
1871         return(FAILURE);
1872
1873     temp_ptr->device_info_string = (char *)
1874     DREALLOC((unsigned)(strlen(name_ptr) + 1));
1875
1876     if (temp_ptr->device_info_string == NULL) {
1877         DFREE((char *)temp_ptr);
1878         return(FAILURE);
1879     }
1880
1881     (void)strcpy(temp_ptr->device_info_string, name_ptr);
1882
1883     temp_ptr->definition = def_ptr;
1884     extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
1885
1886     temp_ptr->dab_info_index = dab_info_index;
1887     temp_ptr->failed_to_alloc_ptrn = FALSE;
1888     temp_ptr->fatal_error = FALSE;
1889     temp_ptr->had_aborted_pin = FALSE;
1890
1891     temp_ptr->restore_state = SENT_RECV_NOTHING;
1892
1893     dab_ptr = dab_list[temp_ptr->dab_info_index];
1894
1895     temp_ptr->pin_info_table =
1896     (PIN_INFO *)DREALLOC((unsigned)(def_ptr->pin_cat * sizeof(PIN_INFO)));
1897     if (temp_ptr->pin_info_table == NULL)
1898         alloc_error = TRUE;
1899     else {
1900         /* Firstly, clear the entire pin_info_table. */
1901         bzero((char *)temp_ptr->pin_info_table,
1902              (int)(def_ptr->pin_cat * sizeof(PIN_INFO)));
1903         /* Secondly, initialize each element of the pin_info_table */
1904         for (pinno = def_ptr->pin_cat - 1; pinno >= 0; --pinno) {
1905             pin_spec_ptr = extra_def_ptr->pin_table[pinno];
1906             pin = temp_ptr->pin_info_table[pinno];
1907
1908             pin->old_raw = LOGIC_U;
1909             pin->old_filtered = LOGIC_U;
1910             pin->new_raw = LOGIC_U;
1911             pin->new_filtered = LOGIC_U;
1912             pin->mis_delay = -1;
1913             pin->sim_time = 0;
1914             pin->next_input_pin_index = -1;
1915             pin->next_output_pin_index = -1;
1916             pin->input_pin_is_linked = FALSE;
1917             pin->output_pin_is_linked = FALSE;
1918             pin->delay_type = DELAY_TABLE;
1919             pin->uninitialized_pin = TRUE;
1920             pin->has_resistor = NO_PULLUP_OR_PULLDOWN;

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/util.c | DATE 5/23/89 | PAGE # 17/167 |
|--|---|---------------------------------|-----------------|------------------|
| LINE # | SOURCE TEXT | | | |
| 1921 | pin->direction = pin_spec_ptr->direction; | | | |
| 1922 | | | | |
| 1923 | if ((pin_spec_ptr->direction == IN) | | | |
| 1924 | (pin_spec_ptr->direction == OUT) | | | |
| 1925 | (pin_spec_ptr->direction == IO)) { | | | |
| 1926 | | | | |
| 1927 | if (pin_spec_ptr->pulled_up == 1) { | | | |
| 1928 | pin->has_resistor = PULLUP; | | | |
| 1929 | } | | | |
| 1930 | | | | |
| 1931 | if (pin_spec_ptr->pulled_down == 1) { | | | |
| 1932 | pin->has_resistor = PULLDOWN; | | | |
| 1933 | } | | | |
| 1934 | } | | | |
| 1935 | | | | |
| 1936 | | | | |
| 1937 | | | | |
| 1938 | temp_ptr->first_data_pin_index = -1; | | | |
| 1939 | temp_ptr->first_eval_pin_index = -1; | | | |
| 1940 | temp_ptr->first_store_pin_index = -1; | | | |
| 1941 | | | | |
| 1942 | temp_ptr->is_fault = FALSE; | | | |
| 1943 | temp_ptr->enable_timing_meas = FALSE; | | | |
| 1944 | temp_ptr->use_2_bit_per_pin = FALSE; | | | |
| 1945 | temp_ptr->fault_count = 0; | | | |
| 1946 | temp_ptr->check_input_2_count = MAX_CHECK_INPUT_2_COUNT; | | | |
| 1947 | temp_ptr->sample_count = DEFAULT_SAMPLE_COUNT; | | | |
| 1948 | temp_ptr->evaluation_count = DEFAULT_EVALUATION_COUNT; | | | |
| 1949 | | | | |
| 1950 | temp_ptr->first_eval = TRUE; | | | |
| 1951 | | | | |
| 1952 | temp_ptr->ptrs_loaded = | | | |
| 1953 | (PTRN_BITS *)DCALLOC((unsigned)unit_count, | | | |
| 1954 | (PTRN_BITS *)sizeof(PTRN_BITS)); | | | |
| 1955 | if (temp_ptr->ptrs_loaded == NULL) | | | |
| 1956 | alloc_error = TRUE; | | | |
| 1957 | else { | | | |
| 1958 | /* Initialize the ptrs_loaded to all 1 */ | | | |
| 1959 | for (unitno = 0; unitno < unit_count; ++unitno) { | | | |
| 1960 | unit_ptr = (PTRN_BITS *)temp_ptr->ptrs_loaded[unitno]; | | | |
| 1961 | unit_ptr->word[0] = 0xffffffff; | | | |
| 1962 | unit_ptr->word[1] = 0xffffffff; | | | |
| 1963 | unit_ptr->word[2] = 0xffffffff; | | | |
| 1964 | } | | | |
| 1965 | | | | |
| 1966 | /* Setup the pattern control bits */ | | | |
| 1967 | ptrs_bits_ptr = temp_ptr->ptrs_loaded; | | | |
| 1968 | for (unitno = 0; unitno < unit_count; ++unitno) { | | | |
| 1969 | | | | |
| 1970 | if (dab_ptr->unit_location[unitno].last_in_lane) | | | |
| 1971 | set_pel_ctl(ptrs_bits_ptr->ctl, PEL_CTL_LOAD_DATA_LAST_UNIT); | | | |
| 1972 | else | | | |
| 1973 | set_pel_ctl(ptrs_bits_ptr->ctl, PEL_CTL_LOAD_DATA); | | | |
| 1974 | | | | |
| 1975 | set_pel_sel(ptrs_bits_ptr->ctl, | | | |
| 1976 | dab_ptr->unit_location[unitno].slot_no); | | | |
| 1977 | | | | |
| 1978 | /* Disable the R1 clocks. | | | |
| 1979 | /* Note that the R1 clocks are already disabled because we initialize | | | |
| 1980 | ptrs_loaded to 0. | | | |
| 1981 | */ | | | |
| 1982 | for (wordno = 0; wordno < 5; ++wordno) { | | | |
| 1983 | ptrs_bits_ptr->word[wordno] = | | | |
| 1984 | ((PTRN_BITS *)extra_def_ptr->ident_R1[unitno])->word[wordno]; | | | |
| 1985 | } | | | |
| 1986 | | | | |
| 1987 | ++ptrs_bits_ptr; | | | |
| 1988 | } | | | |
| 1989 | | | | |
| 1990 | | | | |
| 1991 | /* HWDENB_LOADED, LITCHEN_LOADED, LATCHEN_LOADED are initialized in | | | |
| 1992 | load_preamble(). | | | |
| 1993 | */ | | | |
| 1994 | temp_ptr->hwdenb_loaded = | | | |
| 1995 | (PTRN_BITS *)DCALLOC((unsigned)unit_count * sizeof(PTRN_BITS)); | | | |
| 1996 | if (temp_ptr->hwdenb_loaded == NULL) | | | |
| 1997 | alloc_error = TRUE; | | | |
| 1998 | | | | |
| 1999 | temp_ptr->lcyhdb_loaded = | | | |
| 2000 | (PTRN_BITS *)DCALLOC((unsigned)unit_count * sizeof(PTRN_BITS)); | | | |
| 2001 | if (temp_ptr->lcyhdb_loaded == NULL) | | | |
| 2002 | alloc_error = TRUE; | | | |
| 2003 | | | | |
| 2004 | temp_ptr->lcyddb_loaded = | | | |
| 2005 | (PTRN_BITS *)DCALLOC((unsigned)unit_count * sizeof(PTRN_BITS)); | | | |
| 2006 | if (temp_ptr->lcyddb_loaded == NULL) | | | |
| 2007 | alloc_error = TRUE; | | | |
| 2008 | | | | |
| 2009 | temp_ptr->last_consistent_set = | | | |
| 2010 | (PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS)); | | | |
| 2011 | if (temp_ptr->last_consistent_set == NULL) | | | |
| 2012 | alloc_error = TRUE; | | | |
| 2013 | | | | |
| 2014 | temp_ptr->sim_pin_value.data = | | | |
| 2015 | (PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS)); | | | |
| 2016 | if (temp_ptr->sim_pin_value.data == NULL) | | | |
| 2017 | alloc_error = TRUE; | | | |
| 2018 | | | | |
| 2019 | temp_ptr->sim_pin_value.hiz = | | | |
| 2020 | (PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS)); | | | |
| 2021 | if (temp_ptr->sim_pin_value.hiz == NULL) | | | |
| 2022 | alloc_error = TRUE; | | | |
| 2023 | | | | |
| 2024 | temp_ptr->sim_pin_value.unknown = | | | |
| 2025 | (PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS)); | | | |
| 2026 | if (temp_ptr->sim_pin_value.unknown == NULL) | | | |
| 2027 | alloc_error = TRUE; | | | |
| 2028 | | | | |
| 2029 | temp_ptr->sim_pin_value.soft = | | | |
| 2030 | (PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS)); | | | |
| 2031 | if (temp_ptr->sim_pin_value.soft == NULL) | | | |
| 2032 | alloc_error = TRUE; | | | |
| 2033 | | | | |
| 2034 | temp_ptr->last_sample_value.data = | | | |
| 2035 | (PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS)); | | | |
| 2036 | if (temp_ptr->last_sample_value.data == NULL) | | | |
| 2037 | alloc_error = TRUE; | | | |
| 2038 | | | | |
| 2039 | temp_ptr->last_sample_value.hiz = | | | |
| 2040 | (PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS)); | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89
TIME 6:14:53 pm
PAGE # 18/168

```

LINE # SOURCE TEXT
2041 if (temp_ptr->last_sample_value.hiz == NULL)
2042     alloc_error = TRUE;
2043
2044 temp_ptr->last_sample_value.unknowns =
2045     (PTRN_BITS * DALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS)));
2046 if (temp_ptr->last_sample_value.unknowns == NULL)
2047     alloc_error = TRUE;
2048
2049 temp_ptr->last_sample_value.soft =
2050     (PTRN_BITS * DALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS)));
2051 if (temp_ptr->last_sample_value.soft == NULL)
2052     alloc_error = TRUE;
2053
2054 if (user->free_inst_id == NULL)
2055     if (extended_inst_table(user) == FAILURE)
2056         alloc_error = TRUE;
2057
2058 if (alloc_error == TRUE) {
2059     DFREE((char *)temp_ptr->device_info_string);
2060     DFREE((char *)temp_ptr->pin_info_table);
2061     DFREE((char *)temp_ptr->last_consistent_set);
2062     DFREE((char *)temp_ptr->ptrn_loaded);
2063     DFREE((char *)temp_ptr->lcyddb_loaded);
2064     DFREE((char *)temp_ptr->lcyddb_loaded);
2065     DFREE((char *)temp_ptr->sim_pin_value.data);
2066     DFREE((char *)temp_ptr->sim_pin_value.hiz);
2067     DFREE((char *)temp_ptr->sim_pin_value.unknowns);
2068     DFREE((char *)temp_ptr->sim_pin_value.soft);
2069     DFREE((char *)temp_ptr->last_sample_value.data);
2070     DFREE((char *)temp_ptr->last_sample_value.hiz);
2071     DFREE((char *)temp_ptr->last_sample_value.unknowns);
2072     DFREE((char *)temp_ptr->last_sample_value.soft);
2073     return(FAILURE);
2074 }
2075
2076 slot_ptr = user->free_inst_id;
2077
2078 *inst_id_table_index =
2079     (u_short)((u_long)slot_ptr - (u_long)user->instance) /
2080     sizeof(INSTANCE_INFO *);
2081
2082 user->free_inst_id = (INSTANCE_INFO *)slot_ptr;
2083 *slot_ptr = temp_ptr;
2084
2085 return(SUCCESS);
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:53 pm | 19/169 |

```

2161
2162
2163 /* Initialize pin info table */
2164 for (pinno = def_ptr->pin_cat - 1; pinno >= 0; --pinno) {
2165     pin = (instance->pin_info_table)[pinno];
2166     pin->old_raw = LOGIC_U;
2167     pin->old_filtered = LOGIC_U;
2168     pin->new_raw = LOGIC_U;
2169     pin->new_filtered = LOGIC_U;
2170     pin->pin_delay = -1;
2171     pin->pin_time = 0;
2172     pin->next_input_pin_index = -1;
2173     pin->next_output_pin_index = -1;
2174     pin->input_pin_is_linked = FALSE;
2175     pin->output_pin_is_linked = FALSE;
2176     pin->delay_type = DELAY_TABLE;
2177     pin->uninitialized_pin = TRUE;
2178 }
2179 instance->first_data_pin_index = -1;
2180 instance->first_oval_pin_index = -1;
2181 instance->first_start_pin_index = -1;
2182
2183 instance->is_fault = FALSE;
2184 instance->enable_timing_warnings = FALSE;
2185 instance->use_2_bit_per_pin = FALSE;
2186 instance->fault_count = 0;
2187 instance->check_input_z_count = MAX_CHECK_INPUT_Z_COUNT;
2188 instance->sample_count = DEFAULT_SAMPLE_COUNT;
2189 instance->evaluation_count = DEFAULT_EVALUATION_COUNT;
2190
2191 instance->first_oval = TRUE;
2192
2193 /* Initialize the ptrs loaded to all 1 */
2194 for (unitno = 0; unitno < unit_count; ++unitno) {
2195     unit_ptrn = (PTRN_BITS_LONGWORD *) (instance->ptrn_loaded[unitno]);
2196     unit_ptrn->word[0] = 0xffffffff;
2197     unit_ptrn->word[1] = 0xffffffff;
2198     unit_ptrn->word[2] = 0xffffffff;
2199 }
2200
2201 /* Setup the path control bits */
2202 ptrn_bits_ptr = instance->ptrn_loaded;
2203 for (unitno = 0; unitno < unit_count; ++unitno) {
2204     if (def_ptr->unit_location[unitno].last_in_lane)
2205         set_ptrn_ctl(ptrn_bits_ptr->ctl, PCL_CTL_LOAD_DATA_LAST_UNIT);
2206     else
2207         set_ptrn_ctl(ptrn_bits_ptr->ctl, PCL_CTL_LOAD_DATA);
2208
2209     set_ptrn_sel(ptrn_bits_ptr->ctl,
2210                 def_ptr->unit_location[unitno].slot_no);
2211
2212     /* Disable the R1 clocks.
2213      * Note that the R1 clocks are already disabled because we initialize
2214      * ptrn_loaded to 0.
2215      */
2216     for (wordno = 0; wordno < 5; ++wordno) {
2217         ptrn_bits_ptr->word[wordno] |=
2218             ((PTRN_BITS *) (extra_def_ptr->ident_R1[unitno]))->word[wordno];
2219     }
2220     ++ptrn_bits_ptr;
2221 }
2222
2223 /* BUSES LOADED, LATCHES LOADED, LATCHES LOADED are initialized in
2224  * load_preamble().
2225  */
2226
2227 /* Initialize the following PTRN_BITS to 0 */
2228 last_consistent_set_ptr = (PTRN_BITS_LONGWORD *)
2229     (instance->last_consistent_set);
2230 sim_pin_value_data_ptr = (PTRN_BITS_LONGWORD *)
2231     (instance->sim_pin_value.data);
2232 sim_pin_value_hiz_ptr = (PTRN_BITS_LONGWORD *)
2233     (instance->sim_pin_value.hiz);
2234 sim_pin_value_unknown_ptr = (PTRN_BITS_LONGWORD *)
2235     (instance->sim_pin_value.unknown);
2236 sim_pin_value_soft_ptr = (PTRN_BITS_LONGWORD *)
2237     (instance->sim_pin_value.soft);
2238 last_sample_value_data_ptr = (PTRN_BITS_LONGWORD *)
2239     (instance->last_sample_value.data);
2240 last_sample_value_hiz_ptr = (PTRN_BITS_LONGWORD *)
2241     (instance->last_sample_value.hiz);
2242 last_sample_value_unknown_ptr = (PTRN_BITS_LONGWORD *)
2243     (instance->last_sample_value.unknown);
2244 last_sample_value_soft_ptr = (PTRN_BITS_LONGWORD *)
2245     (instance->last_sample_value.soft);
2246
2247 for (unitno = 0; unitno < unit_count; ++unitno) {
2248     for (wordno = 0; wordno < 3; ++wordno) {
2249         last_consistent_set_ptr->word[wordno] = 0;
2250
2251         sim_pin_value_data_ptr->word[wordno] = 0;
2252         sim_pin_value_hiz_ptr->word[wordno] = 0;
2253         sim_pin_value_unknown_ptr->word[wordno] = 0;
2254         sim_pin_value_soft_ptr->word[wordno] = 0;
2255
2256         last_sample_value_data_ptr->word[wordno] = 0;
2257         last_sample_value_hiz_ptr->word[wordno] = 0;
2258         last_sample_value_unknown_ptr->word[wordno] = 0;
2259         last_sample_value_soft_ptr->word[wordno] = 0;
2260     }
2261 }
2262
2263 ++last_consistent_set_ptr;
2264 ++sim_pin_value_data_ptr;
2265 ++sim_pin_value_hiz_ptr;
2266 ++sim_pin_value_unknown_ptr;
2267 ++sim_pin_value_soft_ptr;
2268
2269 ++last_sample_value_data_ptr;
2270 ++last_sample_value_hiz_ptr;
2271 ++last_sample_value_unknown_ptr;
2272 ++last_sample_value_soft_ptr;
2273
2274
2275
2276
2277
2278 rls_instance(user, inst_id_table_index);
2279 USER_INFO user;
2280 u_short inst_id_table_index;

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:53 pm | 20/170 |

```

2281 |
2282 |
2283 |     INSTANCE_INFO    =instance;
2284 |
2285 |     instance = user->instance/"-- id_table_index);
2286 |
2287 |     DFREE((char *)instance->device_info_string);
2288 |
2289 |     DFREE((char *)instance->pin_info_table);
2290 |
2291 |     DFREE((char *)instance->last_consistent_set);
2292 |
2293 |     DFREE((char *)instance->ptrs_loaded);
2294 |     DFREE((char *)instance->pin_ptr_value_loaded);
2295 |     DFREE((char *)instance->keycdb_loaded);
2296 |     DFREE((char *)instance->keycdb_loaded);
2297 |
2298 |     DFREE((char *)instance->pin_ptr_value.data);
2299 |     DFREE((char *)instance->pin_ptr_value.hiz);
2300 |     DFREE((char *)instance->pin_ptr_value.unknown);
2301 |     DFREE((char *)instance->pin_ptr_value.soft);
2302 |
2303 |     DFREE((char *)instance->last_sample_value.data);
2304 |     DFREE((char *)instance->last_sample_value.hiz);
2305 |     DFREE((char *)instance->last_sample_value.unknown);
2306 |     DFREE((char *)instance->last_sample_value.soft);
2307 |
2308 |     DFREE((char *)instance);
2309 |
2310 |     user->instance(inst_id_table_index) = (INSTANCE_INFO *)user->free_inst_id;
2311 |     user->free_inst_id = instance(inst_id_table_index);
2312 |
2313 | }
2314 |
2315 | /* Definition of user, def_id_table_index */
2316 | USER_INFO *user;
2317 | u_short def_id_table_index;
2318 | {
2319 |     EXTRA_DEVICE_SPEC *extra_def_ptr;
2320 |     DEVICE_SPEC *def_ptr;
2321 |
2322 |     def_ptr = user->definition(def_id_table_index);
2323 |
2324 |     extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
2325 |     DFREE((char *)extra_def_ptr->ident_inputs);
2326 |     DFREE((char *)extra_def_ptr->ident_outputs);
2327 |     DFREE((char *)extra_def_ptr->ident_ios);
2328 |     DFREE((char *)extra_def_ptr->ident_xi);
2329 |     DFREE((char *)extra_def_ptr->ident_xi);
2330 |     DFREE((char *)extra_def_ptr->ident_store);
2331 |
2332 |     DFREE((char *)extra_def_ptr);
2333 |
2334 |     /* Free the DEVICE_SPEC structure by calling a procedure */
2335 |     free_device(def_ptr);
2336 |
2337 |     /* Link the free list */
2338 |     user->definition(def_id_table_index) = (DEVICE_SPEC *)user->free_def_id;
2339 |     user->free_def_id = instance(def_id_table_index);
2340 | }
2341 |
2342 | set_seq_end_bit(instance, lane_addr_info, alloc_tmp_block, seq_end_addr,
2343 | inst_block_number)
2344 | {
2345 |     INSTANCE_INFO *instance;
2346 |     LANE_ADDR_INFO *lane_addr_info;
2347 |     u_char alloc_tmp_block;
2348 |     u_long seq_end_addr;
2349 |     u_long inst_block_number;
2350 |
2351 |     /* The address of the last unit is specified in "lane_addr_info".
2352 |     * If "alloc_tmp_block" = FALSE then we should look at
2353 |     * instance->lane_addr[]_prev_max_addr if we need to set the
2354 |     * SEQ_END_BIT in the previous block, otherwise we should look at
2355 |     * instance->last_addr[]_max_addr.
2356 |     */
2357 |
2358 |     DAB_INFO *dab_ptr;
2359 |     u_long addr;
2360 |     u_long link_table_addr;
2361 |     u_long temp;
2362 |     u_short fault_block_number;
2363 |     u_short ptrs_count;
2364 |     u_char lane0;
2365 |     u_char i;
2366 |
2367 |     dab_ptr = dab_list(instance->dab_info_index);
2368 |
2369 |     /* Fix the block link if it is a fault pattern sequence */
2370 |     if (instance->is_fault == TRUE) {
2371 |         if (instance->disjoint_flag == FALSE) {
2372 |             for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
2373 |                 if (!dab_ptr->lane_used[lane0])
2374 |                     continue;
2375 |
2376 |                 link_table_addr =
2377 |                     ptol(instance->last_common_block_addr[lane0]);
2378 |
2379 |                 inst_block_number[lane0] =
2380 |                     read_loc_long((u_long *)link_table_addr) & BLOCK_NUMBER_MASK;
2381 |
2382 |                 fault_block_number = ptob(instance->var_seq_addr[lane0]);
2383 |
2384 |                 if (instance->last_common_block_is_feedback) {
2385 |                     for (i = 0; i < instance->fb_block_size[lane0]; ++i) {
2386 |                         write_loc_long((u_long *)link_table_addr,
2387 |                             (u_long)fault_block_number);
2388 |                         link_table_addr += LINK_TABLE_ADDR_INC;
2389 |                     }
2390 |                 }
2391 |                 else {
2392 |                     write_loc_long((u_long *)link_table_addr,
2393 |                         (u_long)fault_block_number);
2394 |                 }
2395 |             }
2396 |         }
2397 |     }
2398 |
2399 |     for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
2400 |         if (dab_ptr->lane_used[lane0]) {

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/util.c | DATE 5/23/89 | PAGE # 21/171 |
|--|--|---|-----------------|------------------|
| LINE # | | SOURCE TEXT | | |
| 2401 | | addr = lane_addr[instance].last_unit_addr; | | |
| 2402 | | ptrs_count = PTRN_COUNT_ON_BLOCK(addr); | | |
| 2403 | | if (ptrs_count <= SEQ_END_LATENCY) { | | |
| 2404 | | if (alloc_temp_block == TRUE) { | | |
| 2405 | | addr = instance->lane_addr[instance].max_addr - | | |
| 2406 | | (SEQ_END_LATENCY + 1 - ptrs_count) * PTRN_ADDR_INC + | | |
| 2407 | | LANE_SEGMENT_C_OFFSET; | | |
| 2408 | | } | | |
| 2409 | | else { | | |
| 2410 | | addr = instance->lane_addr[instance].prev_max_addr - | | |
| 2411 | | (SEQ_END_LATENCY + 1 - ptrs_count) * PTRN_ADDR_INC + | | |
| 2412 | | LANE_SEGMENT_C_OFFSET; | | |
| 2413 | | } | | |
| 2414 | | /* The SEQ_END bit should be set in the current block */ | | |
| 2415 | | addr = addr - SEQ_END_LATENCY * PTRN_ADDR_INC + | | |
| 2416 | | LANE_SEGMENT_C_OFFSET; | | |
| 2417 | | } | | |
| 2418 | | seq_end_addr[instance] = addr; | | |
| 2419 | | DPRINTF(("setting SEQ_END bit at addr: %08x\n", addr)); | | |
| 2420 | | temp = read_loc_long((u_long *)addr); | | |
| 2421 | | write_loc_long((u_long *)addr, (u_long)(temp FEL_STOP_MASK)); | | |
| 2422 | | } | | |
| 2423 | | } | | |
| 2424 | | remove_seq_end_bit(instance, seq_end_addr, inst_block_number); | | |
| 2425 | | INSTANCE_INFO *instance; | | |
| 2426 | | u_long seq_end_addr; | | |
| 2427 | | u_long inst_block_number; | | |
| 2428 | | DAB_INFO *dab_ptr; | | |
| 2429 | | u_long temp; | | |
| 2430 | | u_long link_table_addr; | | |
| 2431 | | u_char lane_no; | | |
| 2432 | | u_char i; | | |
| 2433 | | dab_ptr = dab_list[instance->dab_info_index]; | | |
| 2434 | | for (lane_no = 0; lane_no < MAX_LANE_COUNT; ++lane_no) { | | |
| 2435 | | if (dab_ptr->lane_used[lane_no]) { | | |
| 2436 | | DPRINTF(("removing seq end bit addr: %08x\n", seq_end_addr[instance])); | | |
| 2437 | | temp = read_loc_long((u_long *)seq_end_addr[instance]); | | |
| 2438 | | write_loc_long((u_long *)seq_end_addr[instance], | | |
| 2439 | | (u_long)(temp & FEL_STOP_MASK)); | | |
| 2440 | | } | | |
| 2441 | | /* Restore the block link if it is a fault pattern sequence */ | | |
| 2442 | | if (instance->is_fault == TRUE) { | | |
| 2443 | | if (instance->disjoint_flag == FALSE) { | | |
| 2444 | | for (lane_no = 0; lane_no < MAX_LANE_COUNT; ++lane_no) { | | |
| 2445 | | if (!dab_ptr->lane_used[lane_no]) | | |
| 2446 | | continue; | | |
| 2447 | | link_table_addr = | | |
| 2448 | | ptol(instance->last_common_block_addr[lane_no]); | | |
| 2449 | | if (instance->last_common_block_is_feedback) { | | |
| 2450 | | for (i = 0; i < instance->fb_block_size[lane_no]; ++i) { | | |
| 2451 | | write_loc_long((u_long *)link_table_addr, | | |
| 2452 | | (u_long)inst_block_number[lane_no]); | | |
| 2453 | | link_table_addr += LINK_TABLE_ADDR_INC; | | |
| 2454 | | } | | |
| 2455 | | else { | | |
| 2456 | | write_loc_long((u_long *)link_table_addr, | | |
| 2457 | | (u_long)inst_block_number[lane_no]); | | |
| 2458 | | } | | |
| 2459 | | } | | |
| 2460 | | } | | |
| 2461 | | play_ptr_seq(instance, timeout, changed_dac) | | |
| 2462 | | INSTANCE_INFO *instance; | | |
| 2463 | | u_long timeout; | | |
| 2464 | | u_char changed_dac; | | |
| 2465 | | { | | |
| 2466 | | EXTRA_DEVICE_SPEC *extra_def_ptr; | | |
| 2467 | | u_long start_time; | | |
| 2468 | | u_long elapsed; | | |
| 2469 | | u_char replay_pattern = FALSE; | | |
| 2470 | | extra_def_ptr = (EXTRA_DEVICE_SPEC *)instance->definition->extra_data; | | |
| 2471 | | load_starting_address(instance); | | |
| 2472 | | #ifdef DBASE | | |
| 2473 | | if (lm_tmg_check_locked() != SUCCESS) { | | |
| 2474 | | start_time = lm_time(); | | |
| 2475 | | while (lm_tmg_check_locked() != SUCCESS) { | | |
| 2476 | | if ((lm_time() - start_time) > TMC_LOCK_TIMEOUT) { | | |
| 2477 | | lm_queue_message(ERROR_MSG, "failed to lock Timing Generator"); | | |
| 2478 | | instance->fatal_error = TRUE; | | |
| 2479 | | return(FAILURE); | | |
| 2480 | | } | | |
| 2481 | | } | | |
| 2482 | | #endif | | |
| 2483 | | /* Check if the DAB configuration has changed. */ | | |
| 2484 | | if (modeler_error.error == TRUE) { | | |
| 2485 | | modeler_error.error = FALSE; | | |
| 2486 | | if ((modeler_error.pac_lane_errors != 0) | | |
| 2487 | | (modeler_error.tmg_error == TRUE) | | |
| 2488 | | (modeler_error.unknown_source_of_interrupt == TRUE)) { | | |
| 2489 | | fatal hardware_error_encountered = TRUE; | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89
TIME 6:14:53 pm

PAGE #
22/172

```

2521  get_fatal_hardware_message();
2522  instance->fatal_error = TRUE;
2523  return(FAILURE);
2524  }
2525
2526  (modeler_error.dab_change) {
2527    reconfigure_dab();
2528  }
2529  else {
2530    /* Check PEL errors */
2531    if (modeler_error.pel_error_list != 0) {
2532      if (check_pel_errors(instance) == FAILURE) {
2533        instance->fatal_error = TRUE;
2534        return(FAILURE);
2535      }
2536    }
2537  }
2538
2539  if (extra_def_ptr->dab_ok == FALSE) {
2540    lm_queue_message(ERROR_MSG, "Device Adapter was removed");
2541    instance->fatal_error = TRUE;
2542    return(FAILURE);
2543  }
2544
2545  if ("changed_dac" == TRUE) {
2546    while (read_timer1() == FAILURE)
2547      "changed_dac" = FALSE;
2548  }
2549
2550  while (read_timer0(telapsed) == FAILURE)
2551    ;
2552
2553  #ifdef DRAKE
2554  /* Start pattern play */
2555  if (lm_tmg_initiate_play() == FAILURE) {
2556    lm_queue_message(ERROR_MSG, "failed to initiate pattern play");
2557    instance->fatal_error = TRUE;
2558    return(FAILURE);
2559  }
2560  #endif
2561
2562  #ifdef MODELER
2563  if (lm_tmg_complete_play(timeout) == FAILURE) {
2564    lm_queue_message(ERROR_MSG, "failed to complete pattern play");
2565    instance->fatal_error = TRUE;
2566    return(FAILURE);
2567  }
2568  #endif
2569
2570  #ifdef DIRECTOON
2571  /* Poll CPU status for TMC interrupt */
2572  start_time = lm_time();
2573  while (cpu_tmg_interrupt_status() == FALSE) {
2574    if ((lm_time() - start_time) > timeout) {
2575      DPRINTF(("play_ptrn_seq: TIMEOUT\n"));
2576      lm_tmg_abort_play();
2577      DPRINTF(("finished aborting play\n"));
2578      tmoptr->pattern_intr_enable = 0;
2579      lm_queue_message(ERROR_MSG, "timeout during pattern play");
2580      instance->fatal_error = TRUE;
2581      return(FAILURE);
2582    }
2583  }
2584  tmoptr->pattern_intr_enable = 0;
2585  #else
2586  /* Do nothing if it's data base access */
2587  #endif
2588  #endif
2589
2590  if (modeler_error.error == TRUE) {
2591    modeler_error.error = FALSE;
2592
2593    if ((modeler_error.pac_lase_errors != 0) ||
2594        (modeler_error.tmg_error == TRUE) ||
2595        (modeler_error.unknown_source_of_interrupt == TRUE)) {
2596      fatal_hardware_error_accounted = TRUE;
2597      get_fatal_hardware_message();
2598      instance->fatal_error = TRUE;
2599      return(FAILURE);
2600    }
2601  }
2602
2603  if (modeler_error.dab_change) {
2604    reconfigure_dab();
2605    replay_pattern = TRUE;
2606  }
2607  else {
2608    /* Check PEL errors */
2609    if (modeler_error.pel_error_list != 0) {
2610      if (check_pel_errors(instance) == FAILURE) {
2611        instance->fatal_error = TRUE;
2612        return(FAILURE);
2613      }
2614    }
2615  }
2616
2617  if (extra_def_ptr->dab_ok == FALSE) {
2618    lm_queue_message(ERROR_MSG, "Device Adapter was removed");
2619    instance->fatal_error = TRUE;
2620    return(FAILURE);
2621  }
2622
2623  if (replay_pattern == TRUE) {
2624    /* The DAB configuration was changed but, this device was not
2625     * invalidated by reconfigure_dab() (i.e. extra_def_ptr->dab_ok is still
2626     * TRUE). This could mean that this particular device is not touched
2627     * at all or that this device was removed but inserted back in.
2628     * Play the pattern again to take care of the second condition.
2629     */
2630    if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
2631      instance->fatal_error = TRUE;
2632      return(FAILURE);
2633    }
2634  }
2635
2636  return(SUCCESS);
2637
2638  }
2639
2640  return_all_ptrn_block(instance);

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:53 pm | 23/173 |

```

LINE # SOURCE TEXT
2641 INSTANCE_INFO *instance;
2642 {
2643     /* Returns all of the pattern block used by this instance. This routine
2644     * looks at the seq_start_addr (for instance) or var_seq_addr (for fault)
2645     * to find the beginning of the pattern block chain.
2646     */
2647
2648     LANE_ADDR_INFO *lane_ptr;
2649     EXTRA_DEVICE_SPEC *extra_def_ptr;
2650     u_long block_addr;
2651     u_short next_block_number;
2652     u_char lanes;
2653     u_char fb_block_count;
2654     char i;
2655
2656     DPRINTF(("Inside returns_all_ptr_block\n"));
2657
2658     extra_def_ptr = (EXTRA_DEVICE_SPEC *)instance->definition->extra_data;
2659
2660     for (lanes = 0; lanes < MAX_LANE_COUNT; ++lanes) {
2661
2662         lane_ptr = (LANE_ADDR_INFO *)instance->lane_addr[lanes];
2663
2664         if (extra_def_ptr->lane_used & (1 << lanes)) {
2665
2666             if (instance->is_fault == TRUE) {
2667                 block_addr = instance->var_seq_addr[lanes];
2668                 instance->var_seq_addr[lanes] = 0;
2669             }
2670             else {
2671                 block_addr = instance->seq_start_addr[lanes];
2672                 instance->seq_start_addr[lanes] = 0;
2673             }
2674
2675             /* This happens if we fail when we call new_block() */
2676             if (block_addr == 0) {
2677                 printf("block_addr is 0\n");
2678                 continue;
2679             }
2680
2681             next_block_number = read_branch_table_content(block_addr);
2682             while (next_block_number != 0) {
2683
2684                 DPRINTF(("next_block_number: %08x\n", next_block_number));
2685                 release_block(block_addr, lanes);
2686
2687                 block_addr = (block_addr & LANE_ADDR_MASK) +
2688                     (next_block_number << BLOCK_NUMBER_SHIFT);
2689
2690                 next_block_number = read_branch_table_content(block_addr);
2691             }
2692             release_block(block_addr, lanes);
2693
2694             /* If the pattern sequence has feedback, release the remaining
2695             * feedback blocks.
2696             */
2697             if ((instance->fb_block_size[lanes] != 0) &&
                (instance->is_fault == FALSE)) {
2698
2699                 DPRINTF(("releasing remaining feedback blocks\n"));
2700                 block_addr = instance->fb_block_addr[lanes] + BLOCK_ADDR_INC;
2701                 fb_block_count = instance->fb_block_size[lanes] - 1;
2702
2703                 instance->fb_block_addr[lanes] = 0;
2704                 instance->fb_block_size[lanes] = 0;
2705
2706                 if (block_addr != BLOCK_ADDR_INC) {
2707                     for (i = 0; i < fb_block_count; ++i) {
2708                         release_block(block_addr, lanes);
2709                         block_addr += BLOCK_ADDR_INC;
2710                     }
2711                 }
2712             }
2713         }
2714     }
2715 }
2716
2717 #ifdef DEBUG
2718 print_pin_changes(instance)
2719 INSTANCE_INFO *instance;
2720 {
2721     PIN_INFO *pin_info;
2722     short pin_number;
2723
2724     DPRINTF(("DATA pin changes: \n"));
2725     pin_number = instance->first_data_pin_index;
2726     while (pin_number != -1) {
2727         pin_info = instance->pin_info_table[pin_number];
2728         DPRINTF((" pin: %d state: %d", pin_number, pin_info->sim_time));
2729         switch (pin_info->old_filtered) {
2730             case LOGIC_0:
2731                 DPRINTF((" logval: 0\n"));
2732                 break;
2733             case LOGIC_1:
2734                 DPRINTF((" logval: 1\n"));
2735                 break;
2736             case LOGIC_20:
2737                 DPRINTF((" logval: 20\n"));
2738                 break;
2739             case LOGIC_21:
2740                 DPRINTF((" logval: 21\n"));
2741                 break;
2742             case LOGIC_U:
2743                 DPRINTF((" logval: U\n"));
2744                 break;
2745             case LOGIC_S0:
2746                 DPRINTF((" logval: S0\n"));
2747                 break;
2748             case LOGIC_S1:
2749                 DPRINTF((" logval: S1\n"));
2750                 break;
2751             default:
2752                 break;
2753         }
2754         pin_number = pin_info->next_input_pin_index;
2755     }
2756 }

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89
TIME 6:14:53 pm

PAGE #
24/174

LINE # SOURCE TEXT

```

2761 }
2762
2763 DPRINTF(("EVAL pin changes: \n"));
2764 pin_number = instance->first_eval_pin_index;
2765 while (pin_number != -1) {
2766     pin_info = instance->pin_info_table[pin_number];
2767     DPRINTF((" SW: %d stime: %d", pin_number, pin_info->sin_time));
2768     switch (pin_info->old_filtered) {
2769     case LOGIC_0:
2770         DPRINTF((" logicval: 0\n"));
2771         break;
2772     case LOGIC_1:
2773         DPRINTF((" logicval: 1\n"));
2774         break;
2775     case LOGIC_20:
2776         DPRINTF((" logicval: 20\n"));
2777         break;
2778     case LOGIC_21:
2779         DPRINTF((" logicval: 21\n"));
2780         break;
2781     case LOGIC_U:
2782         DPRINTF((" logicval: U\n"));
2783         break;
2784     case LOGIC_S0:
2785         DPRINTF((" logicval: S0\n"));
2786         break;
2787     case LOGIC_S1:
2788         DPRINTF((" logicval: S1\n"));
2789         break;
2790     default:
2791         break;
2792     }
2793     pin_number = pin_info->next_input_pin_index;
2794 }
2795
2796 DPRINTF(("STORE pin changes: \n"));
2797 pin_number = instance->first_store_pin_index;
2798 while (pin_number != -1) {
2799     pin_info = instance->pin_info_table[pin_number];
2800     DPRINTF((" SW: %d stime: %d", pin_number, pin_info->sin_time));
2801     switch (pin_info->old_filtered) {
2802     case LOGIC_0:
2803         DPRINTF((" logicval: 0\n"));
2804         break;
2805     case LOGIC_1:
2806         DPRINTF((" logicval: 1\n"));
2807         break;
2808     case LOGIC_20:
2809         DPRINTF((" logicval: 20\n"));
2810         break;
2811     case LOGIC_21:
2812         DPRINTF((" logicval: 21\n"));
2813         break;
2814     case LOGIC_U:
2815         DPRINTF((" logicval: U\n"));
2816         break;
2817     case LOGIC_S0:
2818         DPRINTF((" logicval: S0\n"));
2819         break;
2820     case LOGIC_S1:
2821         DPRINTF((" logicval: S1\n"));
2822         break;
2823     default:
2824         break;
2825     }
2826     pin_number = pin_info->next_input_pin_index;
2827 }
2828 }
2829 #endif
2830
2831 fill_in_extra_data(def_ptr, dab_info_index)
2832 DEVICE_SPEC *def_ptr;
2833 char dab_info_index;
2834 {
2835     PIN_SPEC
2836     EXTRA_DEVICE_SPEC *pin_spec_ptr;
2837     EXTRA_DEVICE_SPEC *extra_def_ptr;
2838     DAB_INFO
2839     DAB_PTR
2840     UWB_OFFSET
2841     u_short pin_count;
2842     u_short pin_number;
2843     u_char unitno;
2844     u_char wordno;
2845     u_char bitno;
2846     u_char lane;
2847
2848     extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
2849     dab_ptr = dab_list[dab_info_index];
2850
2851     for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {
2852         if (dab_ptr->lane_used[lane]) {
2853             extra_def_ptr->lane_used[lane] = 1 << lane;
2854         }
2855     }
2856
2857     extra_def_ptr->has_to_store = FALSE;
2858     extra_def_ptr->ident_inputs = (PTRN_BITS_LONGWORD *)
2859     DCALLOC((unsigned)dab_ptr->unit_count,
2860             (unsigned)sizeof(PTRN_BITS_LONGWORD));
2861     if (extra_def_ptr->ident_inputs == NULL)
2862         return(FAILURE);
2863
2864     extra_def_ptr->ident_outputs = (PTRN_BITS_LONGWORD *)
2865     DCALLOC((unsigned)dab_ptr->unit_count,
2866             (unsigned)sizeof(PTRN_BITS_LONGWORD));
2867     if (extra_def_ptr->ident_outputs == NULL)
2868         return(FAILURE);
2869
2870     extra_def_ptr->ident_ios = (PTRN_BITS_LONGWORD *)
2871     DCALLOC((unsigned)dab_ptr->unit_count,
2872             (unsigned)sizeof(PTRN_BITS_LONGWORD));
2873     if (extra_def_ptr->ident_ios == NULL)
2874         return(FAILURE);
2875
2876     extra_def_ptr->ident_R1 = (PTRN_BITS_LONGWORD *)
2877     DCALLOC((unsigned)dab_ptr->unit_count,
2878             (unsigned)sizeof(PTRN_BITS_LONGWORD));
2879     if (extra_def_ptr->ident_R1 == NULL)
2880         return(FAILURE);

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/util.c | DATE 5/23/89 | PAGE # 25/175 |
|--|---|---------------------------------|-----------------|------------------|
| LINE # | SOURCE TEXT | | | |
| 2881 | extra_def_ptr->ident_R1 = (PTRN_BITS_LONGWORD *) | | | |
| 2882 | DCALLOC((unsigned)dab_ptr->unit_count, | | | |
| 2883 | (unsigned)sizeof(PTRN_BITS_LONGWORD)); | | | |
| 2884 | if (extra_def_ptr->ident_R1 == NULL) | | | |
| 2885 | return(FAILURE); | | | |
| 2886 | | | | |
| 2887 | extra_def_ptr->ident_store = (PTRN_BITS_LONGWORD *) | | | |
| 2888 | DCALLOC((unsigned)dab_ptr->unit_count, | | | |
| 2889 | (unsigned)sizeof(PTRN_BITS_LONGWORD)); | | | |
| 2890 | if (extra_def_ptr->ident_store == NULL) | | | |
| 2891 | return(FAILURE); | | | |
| 2892 | | | | |
| 2893 | pin_count = dab_ptr->pin_cnt; | | | |
| 2894 | pin_spec_ptr = dab_ptr->pin_table[0]; | | | |
| 2895 | for (pin_number = 0; pin_number < pin_count; ++pin_number) { | | | |
| 2896 | | | | |
| 2897 | if ((pin_spec_ptr->direction == NONE) | | | |
| 2898 | (pin_spec_ptr->direction == POWER) | | | |
| 2899 | (pin_spec_ptr->direction == GROUND) | | | |
| 2900 | (pin_spec_ptr->direction == NC)) { | | | |
| 2901 | ++pin_spec_ptr; | | | |
| 2902 | continue; | | | |
| 2903 | | | | |
| 2904 | | | | |
| 2905 | word_ptr = &pin_spec_ptr->word; | | | |
| 2906 | unitno = word_ptr->unitno; | | | |
| 2907 | wordno = word_ptr->wordno; | | | |
| 2908 | bitno = word_ptr->bitno; | | | |
| 2909 | | | | |
| 2910 | if (pin_spec_ptr->pin_class == STORE) { | | | |
| 2911 | set_ptrn_bit(extra_def_ptr->ident_store [unitno], wordno, bitno); | | | |
| 2912 | | | | |
| 2913 | if (pin_spec_ptr->direction == IO) | | | |
| 2914 | extra_def_ptr->has_io_store = TRUE; | | | |
| 2915 | | | | |
| 2916 | | | | |
| 2917 | switch (pin_spec_ptr->direction) { | | | |
| 2918 | case IN: | | | |
| 2919 | set_ptrn_bit(extra_def_ptr->ident_inputs [unitno], wordno, bitno); | | | |
| 2920 | break; | | | |
| 2921 | case OUT: | | | |
| 2922 | set_ptrn_bit(extra_def_ptr->ident_outputs [unitno], wordno, bitno); | | | |
| 2923 | break; | | | |
| 2924 | case IO: | | | |
| 2925 | set_ptrn_bit(extra_def_ptr->ident_ioa [unitno], wordno, bitno); | | | |
| 2926 | break; | | | |
| 2927 | default: | | | |
| 2928 | break; | | | |
| 2929 | | | | |
| 2930 | | | | |
| 2931 | switch (pin_spec_ptr->clk_format) { | | | |
| 2932 | case R1: | | | |
| 2933 | set_ptrn_bit(extra_def_ptr->ident_R1 [unitno], wordno, bitno); | | | |
| 2934 | break; | | | |
| 2935 | case R0: | | | |
| 2936 | set_ptrn_bit(extra_def_ptr->ident_R2 [unitno], wordno, bitno); | | | |
| 2937 | break; | | | |
| 2938 | default: | | | |
| 2939 | break; | | | |
| 2940 | | | | |
| 2941 | | | | |
| 2942 | ++pin_spec_ptr; | | | |
| 2943 | | | | |
| 2944 | | | | |
| 2945 | return(SUCCESS); | | | |
| 2946 | | | | |
| 2947 | | | | |
| 2948 | increment_unit_addr(unit_addr, unit_count, unit_count_per_lane) | | | |
| 2949 | u_long unit_addr[]; | | | |
| 2950 | u_short unit_count; | | | |
| 2951 | u_short unit_count_per_lane; | | | |
| 2952 | { | | | |
| 2953 | u_long next_block_addr; | | | |
| 2954 | u_long max_block_addr; | | | |
| 2955 | u_short next_block_number; | | | |
| 2956 | u_short addr_inc_per_lane; | | | |
| 2957 | u_char unitno; | | | |
| 2958 | | | | |
| 2959 | addr_inc_per_lane = unit_count_per_lane * PTRN_ADDR_INC; | | | |
| 2960 | | | | |
| 2961 | for (unitno = 0; unitno < unit_count; ++unitno) { | | | |
| 2962 | | | | |
| 2963 | max_block_addr = (unit_addr[unitno] & BLOCK_START_MASK) + BLOCK_ADDR_INC; | | | |
| 2964 | | | | |
| 2965 | if ((unit_addr[unitno] + addr_inc_per_lane) >= max_block_addr) { | | | |
| 2966 | | | | |
| 2967 | next_block_number = read_branch_table_content(unit_addr[unitno]); | | | |
| 2968 | | | | |
| 2969 | next_block_addr = (unit_addr[unitno] & LANE_ADDR_MASK) + | | | |
| 2970 | (next_block_number << BLOCK_NUMBER_SHIFT); | | | |
| 2971 | | | | |
| 2972 | unit_addr[unitno] = next_block_addr + | | | |
| 2973 | unit_addr[unitno] + addr_inc_per_lane - max_block_addr; | | | |
| 2974 | | | | |
| 2975 | else { | | | |
| 2976 | unit_addr[unitno] += addr_inc_per_lane; | | | |
| 2977 | | | | |
| 2978 | } | | | |
| 2979 | | | | |
| 2980 | | | | |
| 2981 | set_feedback_branch(instance, branch_offset) | | | |
| 2982 | INSTANCE_INFO *instance; | | | |
| 2983 | short branch_offset; | | | |
| 2984 | { | | | |
| 2985 | | | | |
| 2986 | DEVICE_SPEC *def; | | | |
| 2987 | DAB_INFO *dab_ptr; | | | |
| 2988 | u_long addr; | | | |
| 2989 | u_long temp; | | | |
| 2990 | u_char lanesno; | | | |
| 2991 | | | | |
| 2992 | def = instance->definition; | | | |
| 2993 | dab_ptr = dab_list(instance->dab_info_index); | | | |
| 2994 | | | | |
| 2995 | | | | |
| 2996 | for (lanesno = 0; lanesno < MAX_LANE_COUNT; ++lanesno) { | | | |
| 2997 | | | | |
| 2998 | if (dab_ptr->lane_used[lanesno]) { | | | |
| 2999 | addr = instance->fb_block_addr[lanesno] + | | | |
| 3000 | branch_offset + PTRN_ADDR_INC + LANE_SEGMENT_C_OFFSET; | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89 PAGE #
TIME 6:14:53 pm 26/176

```

LINE # SOURCE TEXT
3001 DPRINTF(("set feedback branch at addr: %08X\n", addr));
3002
3003 temp = read_loc_long((u_long *)addr);
3004
3005 if (def->fb_type == FEEDBACK_RISE)
3006     temp = temp & PEL_BRANCH_MASK | BRANCH_RISE << PEL_BRANCH_SHIFT;
3007 else
3008     temp = temp & PEL_BRANCH_MASK | BRANCH_FALL << PEL_BRANCH_SHIFT;
3009
3010 write_loc_long((u_long *)addr, temp);
3011
3012 }
3013
3014 }
3015
3016
3017
3018 read_magic_full_sample_reg(instance, full_value_ptr)
3019 INSTANCE_INFO *instance,
3020 FULL_VALUE *full_value_ptr,
3021 {
3022
3023 #ifdef DBASE
3024     DAB_INFO *dab_ptr;
3025     u_long addr;
3026     register u_long magic_addr;
3027     u_char total_unit;
3028     u_char unitno;
3029     u_char lane_no;
3030     u_char slotno;
3031     register u_char wordno;
3032     register PTRN_BITS *data_ptr;
3033     register PTRN_BITS *hiz_ptr;
3034     register PTRN_BITS *unk_ptr;
3035
3036     DPRINTF(("inside read_magic_full_sample_reg\n"));
3037
3038     dab_ptr = dab_list(instance->dab_info_index);
3039
3040     data_ptr = full_value_ptr->data;
3041     hiz_ptr = full_value_ptr->hiz;
3042     unk_ptr = full_value_ptr->unknown;
3043
3044     total_unit = dab_ptr->unit_count;
3045     for (unitno = 0; unitno < total_unit; ++unitno) {
3046         lane_no = dab_ptr->unit_location[unitno].lane_no;
3047         slotno = dab_ptr->unit_location[unitno].slot_no;
3048
3049         addr = LANE_0_START_ADDR + lane_no * LANE_ADDR_INC +
3050             LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC;
3051
3052         magic_addr = addr + PEL_MC_0_OFFSET +
3053             (MAX_SHORT_WORD_COUNT - 1) * PEL_MC_ADDR_INC;
3054         for (wordno = 0; wordno < MAX_SHORT_WORD_COUNT; ++wordno) {
3055             /* MAGIC: 0 -> pin 15...8
3056              * ptrn_ptr->word[0] -> pin 77...64
3057              */
3058             data_ptr->word[wordno] = read_loc_long((u_long *)
3059                 (magic_addr + MC_VALUE_SAMPLE_REG_OFFSET));
3060             hiz_ptr->word[wordno] = read_loc_long((u_long *)
3061                 (magic_addr + MC_HIZ_SAMPLE_REG_OFFSET));
3062             unk_ptr->word[wordno] = read_loc_long((u_long *)
3063                 (magic_addr + MC_U_SAMPLE_REG_OFFSET));
3064             magic_addr += PEL_MC_ADDR_INC;
3065         }
3066
3067         DPRINTF((" DATA: %08X %08X %08X\n",
3068             ((PTRN_BITS_LONGWORD *)data_ptr)->word[0],
3069             ((PTRN_BITS_LONGWORD *)data_ptr)->word[1],
3070             ((PTRN_BITS_LONGWORD *)data_ptr)->word[2]));
3071
3072         DPRINTF((" HIZ: %08X %08X %08X\n",
3073             ((PTRN_BITS_LONGWORD *)hiz_ptr)->word[0],
3074             ((PTRN_BITS_LONGWORD *)hiz_ptr)->word[1],
3075             ((PTRN_BITS_LONGWORD *)hiz_ptr)->word[2]));
3076
3077         DPRINTF((" UNK: %08X %08X %08X\n",
3078             ((PTRN_BITS_LONGWORD *)unk_ptr)->word[0],
3079             ((PTRN_BITS_LONGWORD *)unk_ptr)->word[1],
3080             ((PTRN_BITS_LONGWORD *)unk_ptr)->word[2]));
3081
3082         ++data_ptr;
3083         ++hiz_ptr;
3084         ++unk_ptr;
3085     }
3086 #else
3087     DAB_INFO *dab_ptr;
3088     DAB_OFFSET *dab_ptr;
3089     char line[80];
3090     u_long pin_number;
3091     u_short total_pin;
3092     u_char pin_value;
3093     u_char unitno;
3094     u_char wordno;
3095     u_char bitno;
3096
3097     DPRINTF(("inside read_magic_full_sample_reg()\n"));
3098
3099     if (is_measure_delay == TRUE) {
3100         dab_ptr = dab_list(instance->dab_info_index);
3101         total_pin = dab_ptr->unit_count * 80;
3102
3103         for (pin_number = 0; pin_number < total_pin; ++pin_number) {
3104             if (tm_result_array(pin_number).set_by_user == TRUE) {
3105
3106                 unitno = pin_to_short_offset(pin_number);
3107                 wordno = unitno->wordno;
3108                 bitno = unitno->bitno;
3109
3110                 set_pin_value(full_value_ptr, unitno, wordno, bitno,
3111                     tm_result_array(pin_number).steady_state_full_value);
3112             }
3113         }
3114     }
3115 }
3116
3117
3118
3119
3120

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89
TIME 6:14:53 pm

PAGE #
27/177

```

LINE # SOURCE TEXT
3121     return;
3122 }
3123
3124 DPRINTF(("enter starting pin_number: \n"));
3125 gets(line);
3126 sscanf(line, "%d", &pin_number);
3127
3128 /* assume that the MIB pin_number is sequential */
3129
3130 DPRINTF(("enter pin value on at a time, and with 'X' (legal val: 0, 1, 20, 21, U)\n"));
3131
3132 while (1) {
3133     DPRINTF(("pin value: "));
3134     gets(line);
3135     if (strcmp(line, "0") == 0)
3136         pin_value = LOGIC_0;
3137     else if (strcmp(line, "1") == 0)
3138         pin_value = LOGIC_1;
3139     else if (strcmp(line, "20") == 0)
3140         pin_value = LOGIC_20;
3141     else if (strcmp(line, "21") == 0)
3142         pin_value = LOGIC_21;
3143     else if (strcmp(line, "U") == 0)
3144         pin_value = LOGIC_U;
3145     else if (strcmp(line, "X") == 0)
3146         break;
3147     else {
3148         DPRINTF(("illegal value. ignored\n"));
3149         continue;
3150     }
3151
3152     word_ptr = &pin_value;
3153     unitno = word_ptr->unitno;
3154     wordno = word_ptr->wordno;
3155     bitno = word_ptr->bitno;
3156
3157     set_pin_value(full_value_ptr, unitno, wordno, bitno, pin_value);
3158     ++pin_number;
3159 }
3160
3161 #endif
3162
3163 /* ARGUSED */
3164 read_magic_timing_sample_reg(instance, ptrn_long_ptr, current_time)
3165 INSTANCE_INFO *instance,
3166 PTRN_BITS_LONGWORD *ptrn_long_ptr,
3167 u_short current_time;
3168 {
3169     #ifndef DBASE
3170     DAB_INFO *dab_ptr;
3171     u_long addr;
3172     u_long magic_addr;
3173     u_char total_unit;
3174     u_char unitno;
3175     u_char lane0;
3176     u_char slotno;
3177     u_char wordno;
3178     PTRN_BITS *ptrn_ptr;
3179
3180     DPRINTF(("inside read_magic_timing_sample_reg\n"));
3181
3182     dab_ptr = dab_list(instance->dab_info_index);
3183
3184     ptrn_ptr = (PTRN_BITS *)ptrn_long_ptr;
3185     total_unit = dab_ptr->unit_count;
3186     for (unitno = 0; unitno < total_unit; ++unitno) {
3187         lane0 = dab_ptr->unit_location(unitno).lane_no;
3188         slotno = dab_ptr->unit_location(unitno).slot_no;
3189
3190         addr = LANE_0_START_ADDR + lane0 * LANE_ADDR_INC +
3191             LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC;
3192
3193         magic_addr = addr + PEL_MC_0_OFFSET + MC_TIMING_SAMPLE_REC_OFFSET +
3194             (MAX_SHORT_WORD_COUNT - 1) * PEL_MC_ADDR_INC;
3195         for (wordno = 0; wordno < MAX_SHORT_WORD_COUNT; ++wordno) {
3196             /* MAGIC 0 -> pin 15..0
3197              * ptrn_ptr->word[0] -> pin 73..64
3198              */
3199             ptrn_ptr->word[wordno] = read_long((u_long *)magic_addr);
3200             magic_addr += PEL_MC_ADDR_INC;
3201         }
3202
3203         DPRINTF(("DATA: 0001 0002 0003\n"));
3204         ((PTRN_BITS_LONGWORD *)ptrn_ptr)->word[0] = ptrn_ptr->word[0];
3205         ((PTRN_BITS_LONGWORD *)ptrn_ptr)->word[1] = ptrn_ptr->word[1];
3206         ((PTRN_BITS_LONGWORD *)ptrn_ptr)->word[2] = ptrn_ptr->word[2];
3207
3208         ++ptrn_ptr;
3209     }
3210
3211     #else
3212     DAB_INFO *dab_ptr;
3213     u_long *word_ptr;
3214     char line[80];
3215     u_long pin_number;
3216     u_short total_pin;
3217     u_char unitno;
3218     u_char wordno;
3219     u_char bitno;
3220
3221     DPRINTF(("inside read_magic_timing_sample_reg()\n"));
3222
3223     if (in_measure_delay == TRUE) {
3224         dab_ptr = dab_list(instance->dab_info_index);
3225         total_pin = dab_ptr->unit_count * 80;
3226         for (pin_number = 0; pin_number < total_pin; ++pin_number) {
3227             if (cm_result_array(pin_number).set_by_user == TRUE) {
3228                 word_ptr = &pin_number;
3229                 unitno = word_ptr->unitno;
3230                 wordno = word_ptr->wordno;
3231                 bitno = word_ptr->bitno;
3232             }
3233         }
3234     }
3235
3236     #endif
3237 }
3238
3239
3240

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:53 pm | 28/178 |

```

LINE # SOURCE TEXT
3241 if (tm_result_array[pin_number].expected_time <= current_time) {
3242     if (tm_result_array[pin_number].two_state_value == LOGIC_0) {
3243         reset_ptrn_bit(&((PTRN_BITS *)ptrn_long_ptr)(unitno),
3244             wordao, bitao);
3245     }
3246     else {
3247         set_ptrn_bit(&((PTRN_BITS *)ptrn_long_ptr)(unitno),
3248             wordao, bitao);
3249     }
3250 }
3251 else {
3252     if (tm_result_array[pin_number].two_state_value == LOGIC_0) {
3253         set_ptrn_bit(&((PTRN_BITS *)ptrn_long_ptr)(unitno),
3254             wordao, bitao);
3255     }
3256     else {
3257         reset_ptrn_bit(&((PTRN_BITS *)ptrn_long_ptr)(unitno),
3258             wordao, bitao);
3259     }
3260 }
3261 }
3262 }
3263 return;
3264 }
3265
3266 DPRINTF(("enter starting pin number: \n"));
3267 gets(line);
3268 sscanf(line, "%d", &pin_number);
3269
3270 /* assume that the bus pin number is sequential */
3271 DPRINTF(("enter pin value on at a time, and with 'X' (legal val: 0, 1)\n"));
3272
3273 while (1) {
3274     uwb_ptr = tpa_to_short_offset(pin_number);
3275     unitno = uwb_ptr->unitno;
3276     wordao = uwb_ptr->wordao;
3277     bitao = uwb_ptr->bitao;
3278
3279     DPRINTF(("pin value: "));
3280     gets(line);
3281
3282     if (strcmp(line, "0") == 0)
3283         reset_ptrn_bit(&((PTRN_BITS *)ptrn_long_ptr)(unitno), wordao, bitao);
3284     else if (strcmp(line, "1") == 0)
3285         set_ptrn_bit(&((PTRN_BITS *)ptrn_long_ptr)(unitno), wordao, bitao);
3286     else if (strcmp(line, "X") == 0)
3287         break;
3288     else {
3289         DPRINTF(("illegal value. ignored\n"));
3290         continue;
3291     }
3292     ++pin_number;
3293 }
3294
3295 }
3296 #endif
3297
3298
3299
3300
3301 clear_ptrn_bits(ptrn_ptr, total_unit)
3302 char *ptrn_ptr
3303 char *total_unit
3304 {
3305     char *memset();
3306     (void)memset(ptrn_ptr, 0, (int)(total_unit * sizeof(PTRN_BITS)));
3307 }
3308
3309 copy_ptrn_bits(source, dest, total_unit)
3310 char *source
3311 char *dest
3312 u_char total_unit
3313 {
3314     char *memcpy();
3315     (void)memcpy(dest, source, (int)(total_unit * sizeof(PTRN_BITS)));
3316 }
3317
3318
3319
3320
3321 init_tm_pin_info_table(tm_pin_info_table, pin_count)
3322 TM_PIN_INFO *tm_pin_info_table;
3323 u_short pin_count;
3324 {
3325     u_short pinno;
3326     for (pinno = 0; pinno < pin_count; ++pinno) {
3327         tm_pin_info_table[pinno].delay_found = FALSE;
3328         tm_pin_info_table[pinno].delay_value_is_set = FALSE;
3329     }
3330 }
3331
3332
3333
3334
3335 setup_gbl_dummy_ptrn(dab_ptr)
3336 DAB_INFO *dab_ptr;
3337 {
3338     u_char unitno;
3339     u_char total_unit;
3340     total_unit = dab_ptr->unit_count;
3341     for (unitno = 0; unitno < total_unit; ++unitno) {
3342         set_pel_sel(&gbl_dummy_ptrn[unitno].ctl,
3343             dab_ptr->unit_location[unitno].slot_no);
3344     }
3345 }
3346
3347
3348
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3360

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89
TIME 6:14:53 pm

PAGE #
29/179

```

LINE #          SOURCE TEXT
3361      extra_def_ptr->source_reg) != SUCCESS) {
3362      lm_queue_message(ERROR_MSG, "Timing Generator error: set frequency");
3363      return(FAILURE);
3364      }
3365
3366      /* Set the mode to EARLY SAMPLE and the EDGE 7 threshold to 255,
3367      * in order to get a sample pulse.
3368      * These values will be overriden if we are doing timing measurement.
3369      */
3370      extra_def_ptr->edge_setting[6] = 255;
3371
3372      if (lm_tmg_set_sample_trigger_mode(
3373          (u_long)EARLYSAMPLETRIGGERMODE) == FAILURE) {
3374          lm_queue_message(ERROR_MSG, "Timing Generator error: set sample trigger mode");
3375          return(FAILURE);
3376      }
3377      return(SUCCESS);
3378  }
3379
3380  load_starting_address(instance)
3381  INSTANCE_INFO *instance;
3382  {
3383      /* Load the starting pattern address on each lane and
3384      * also load the clock frequency register.
3385      */
3386      EXTRA_DEVICE_SPEC *extra_def_ptr;
3387      DAB_INFO *dab_ptr;
3388      u_long addr;
3389      u_char lane0;
3390      u_short blocknum;
3391      u_char clock_speed_reg;
3392
3393      extra_def_ptr = (EXTRA_DEVICE_SPEC *)instance->definition->extra_data;
3394
3395      if (extra_def_ptr->actual_pay_clock_period < PAC_FREQUENCY_THRESHOLD)
3396          clock_speed_reg = 1;
3397      else
3398          clock_speed_reg = 0;
3399
3400      dab_ptr = dab_list(instance->dab_info_index);
3401
3402      /* Tell the TME which lane we are playing to */
3403      lm_tmg_lane_select(dab_ptr->ident_lane);
3404
3405      for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
3406          if (dab_ptr->lane_used[lane0]) {
3407              blocknum = ptob(instance->seq_start_addr[lane0]);
3408              addr = LANE_0_START_ADDR + lane0 * LANE_ADDR_INC +
3409                  LANE_PAC_START_OFFSET;
3410
3411              /* set the starting address */
3412              write_loc_long((u_long *))(addr + PAC_BRANCH_ADDR_REG_OFFSET),
3413                          (u_long)blocknum);
3414
3415              /* set the clock speed register */
3416              write_loc_long((u_long *))(addr + PAC_CLOCK_SPEED_REG_OFFSET),
3417                          (u_long)clock_speed_reg);
3418          }
3419      }
3420
3421      }
3422
3423      #ifdef MODELER
3424      ac_delay()
3425      {
3426      }
3427      #endif
3428
3429      set_edge_and_sample_setting(extra_def_ptr, resolution, sample_threshold);
3430
3431      EXTRA_DEVICE_SPEC *extra_def_ptr;
3432      u_long resolution;
3433      u_long sample_threshold;
3434
3435      #ifdef BRASE
3436      if (lm_tmg_set_edge_settings(extra_def_ptr->logical_clock_period,
3437          extra_def_ptr->edge_setting) != SUCCESS) {
3438          lm_queue_message(ERROR_MSG, "Timing Generator error: set edge settings");
3439          return(FAILURE);
3440      }
3441
3442      if (lm_tmg_set_rising_sample(resolution, sample_threshold) != SUCCESS) {
3443          lm_queue_message(ERROR_MSG, "Timing Generator error: set rising sample");
3444          return(FAILURE);
3445      }
3446
3447      if (lm_tmg_set_falling_sample(extra_def_ptr->falling_sample_reg) !=
3448          SUCCESS) {
3449          lm_queue_message(ERROR_MSG, "Timing Generator error: set falling sample");
3450          return(FAILURE);
3451      }
3452      }
3453      #endif
3454      setup_timer0(TMG_TIMER_VALUE);
3455      return(SUCCESS);
3456  }
3457
3458  calc_dab_voltage(def_ptr, dab_ptr)
3459  DEVICE_SPEC *def_ptr;
3460  DAB_INFO *dab_ptr;
3461  {
3462      EXTRA_DEVICE_SPEC *extra_def_ptr;
3463      u_short temp;
3464      u_short adjusted_val;
3465      u_short adjusted_vsh;
3466      u_short adjusted_vil;
3467      u_short adjusted_vih;
3468      u_short adjusted_vlth;
3469      u_short adjusted_vsth;
3470
3471      extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
3472
3473      if (dab_ptr->dut_vcc_measured < MIN_DUTVCC) {
3474          lm_queue_message(ERROR_MSG, "device VCC measured (%d mV) is too low, min allowed: %d mV",
3475              dab_ptr->dut_vcc_measured,
3476              MIN_DUTVCC);
3477          return(FAILURE);
3478      }
3479
3480      if (dab_ptr->dut_vcc_measured > MAX_DUTVCC) {

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:53 pm | 30/180 |

```

LINE #          SOURCE TEXT
3481      lm_queue_message(ERROR_MSG, "device VCC measured (%d mV) is too high, max allowed: %d mV",
3482      dab_ptr->dut_vcc_measured,
3483      MAX_DUTVCC);
3484      return(FAILURE);
3485  }
3486
3487  if (within_tolerance(dab_ptr->dut_vcc_measured, def_ptr->vcc) == FALSE) {
3488      lm_queue_message(ERROR_MSG, "device VCC does not match, measured: %d mV specified: %d mV",
3489      dab_ptr->dut_vcc_measured, def_ptr->vcc);
3490      return(FAILURE);
3491  }
3492
3493  if (def_ptr->val > def_ptr->vcc / 5) {
3494      lm_queue_message(ERROR_MSG, "Val out of range. Specified: %d mV range: (%d mV - %d mV)",
3495      def_ptr->val, 0, def_ptr->vcc / 5);
3496      return(FAILURE);
3497  }
3498
3499  if ((def_ptr->vsh < def_ptr->vcc / 2) ||
3500      (def_ptr->vsh > def_ptr->vcc)) {
3501      lm_queue_message(ERROR_MSG, "Vsh out of range. Specified: %d mV range: (%d mV - %d mV)",
3502      def_ptr->vsh,
3503      def_ptr->vcc / 2,
3504      def_ptr->vcc);
3505      return(FAILURE);
3506  }
3507
3508  if (def_ptr->vth > def_ptr->vcc / 5) {
3509      lm_queue_message(ERROR_MSG, "Vth out of range. Specified: %d mV range: (%d mV - %d mV)",
3510      def_ptr->vth, 0, def_ptr->vcc / 5);
3511      return(FAILURE);
3512  }
3513
3514  if ((def_ptr->vth < def_ptr->vcc / 2) ||
3515      (def_ptr->vth > def_ptr->vcc)) {
3516      lm_queue_message(ERROR_MSG, "Vth out of range. Specified: %d mV range: (%d mV - %d mV)",
3517      def_ptr->vth,
3518      def_ptr->vcc / 2,
3519      def_ptr->vcc);
3520      return(FAILURE);
3521  }
3522
3523  if (def_ptr->vil > def_ptr->vcc) {
3524      lm_queue_message(ERROR_MSG, "vil out of range. Specified: %d mV range: (%d mV - %d mV)",
3525      def_ptr->vil,
3526      0,
3527      def_ptr->vcc);
3528      return(FAILURE);
3529  }
3530
3531  if (def_ptr->vih > def_ptr->vcc) {
3532      lm_queue_message(ERROR_MSG, "vih out of range. Specified: %d mV range: (%d mV - %d mV)",
3533      def_ptr->vih,
3534      0,
3535      def_ptr->vcc);
3536      return(FAILURE);
3537  }
3538
3539  adjusted_val = def_ptr->val * dab_ptr->dut_vcc_measured / def_ptr->vcc;
3540  extra_def_ptr->val_dec_value =
3541      REG_VALUE(0, dab_ptr->dut_vcc_measured / 5, adjusted_val);
3542
3543  extra_def_ptr->private_val_dec_value =
3544      REG_VALUE(0, dab_ptr->dut_vcc_measured / 5, 0);
3545
3546  adjusted_vsh = def_ptr->vsh * dab_ptr->dut_vcc_measured / def_ptr->vcc;
3547  extra_def_ptr->vsh_dec_value =
3548      REG_VALUE(dab_ptr->dut_vcc_measured / 2,
3549      dab_ptr->dut_vcc_measured,
3550      adjusted_vsh);
3551
3552  if (def_ptr->vsh + 400 > dab_ptr->dut_vcc_measured)
3553      temp = dab_ptr->dut_vcc_measured;
3554  else
3555      temp = def_ptr->vsh + 400;
3556
3557  extra_def_ptr->private_vsh_dec_value =
3558      REG_VALUE(dab_ptr->dut_vcc_measured / 2,
3559      dab_ptr->dut_vcc_measured,
3560      temp);
3561
3562  adjusted_vth = def_ptr->vth * dab_ptr->dut_vcc_measured / def_ptr->vcc;
3563  extra_def_ptr->vth_dec_value =
3564      REG_VALUE(0, dab_ptr->dut_vcc_measured / 5, adjusted_vth);
3565
3566  adjusted_vhth = def_ptr->vth * dab_ptr->dut_vcc_measured / def_ptr->vcc;
3567  extra_def_ptr->vhth_dec_value =
3568      REG_VALUE(dab_ptr->dut_vcc_measured / 2,
3569      dab_ptr->dut_vcc_measured,
3570      adjusted_vhth);
3571
3572  adjusted_vil = def_ptr->vil * dab_ptr->dut_vcc_measured / def_ptr->vcc;
3573  extra_def_ptr->vil_dec_value =
3574      REG_VALUE(0, dab_ptr->dut_vcc_measured, adjusted_vil);
3575
3576  adjusted_vih = def_ptr->vih * dab_ptr->dut_vcc_measured / def_ptr->vcc;
3577  extra_def_ptr->vih_dec_value =
3578      REG_VALUE(0, dab_ptr->dut_vcc_measured, adjusted_vih);
3579
3580  DPRINTF(("vih : %d mV -> %s", adjusted_vih,
3581      extra_def_ptr->vih_dec_value));
3582  DPRINTF(("vil : %d mV -> %s", adjusted_vil,
3583      extra_def_ptr->vil_dec_value));
3584  DPRINTF(("vth : %d mV -> %s", adjusted_vth,
3585      extra_def_ptr->vth_dec_value));
3586  DPRINTF(("vsh : %d mV -> %s", adjusted_vsh,
3587      extra_def_ptr->vsh_dec_value));
3588  DPRINTF(("vsh : %d mV -> %s", adjusted_vsh,
3589      extra_def_ptr->vsh_dec_value));
3590  DPRINTF(("val : %d mV -> %s", adjusted_val,
3591      extra_def_ptr->val_dec_value));
3592  return(SUCCESS);
3593  }
3594
3595  within_tolerance(ref_value, value)
3596  u_short ref_value;
3597  u_short value;
3598  {
3599      /* Return TRUE if "value" is <= MAX_PERCENT_TOLERANCE percent from
3600      * "ref_value", else return FALSE.

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:53 pm | 31/181 |

```

LINE # SOURCE TEXT
3601 //
3602
3603 if ((value < ref_value - MAX_PERCENT_TOLERANCE * ref_value / 100) ||
3604     (value > ref_value + MAX_PERCENT_TOLERANCE * ref_value / 100))
3605     return SE;;
3606
3607 return(TRUE);
3608 }
3609
3610 verify_soft_drive_current(def_ptr, dab_ptr)
3611 DEVICE_SPEC *def_ptr,
3612 DAB_INFO *dab_ptr,
3613 {
3614     PIN_SPEC *pin_spec_ptr;
3615     u_long total_current;
3616     pin_count;
3617     max_pin_count;
3618     u_short soft_drive_low_count[MAX_LANE_COUNT * MAX_SLOT_COUNT][4];
3619     u_short soft_drive_high_count[MAX_LANE_COUNT * MAX_SLOT_COUNT][4];
3620     pinno;
3621     unitno;
3622     u_char palno;
3623     u_char temp;
3624     u_char current_limit_exceeded = FALSE;
3625
3626     for (palno = 0; palno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++palno) {
3627         for (temp = 0; temp < 4; ++temp) {
3628             soft_drive_low_count[palno][temp] = 0;
3629             soft_drive_high_count[palno][temp] = 0;
3630         }
3631     }
3632
3633     pin_count = def_ptr->pin_cnt;
3634     max_pin_count = dab_ptr->unit_count * MAX_PIN_PER_UNIT;
3635
3636     unitno = ps_to_short_offset(pin_count - 1).unitno;
3637     palno = dab_ptr->unit_location(unitno).lane_no * MAX_SLOT_COUNT +
3638             dab_ptr->unit_location(unitno).slot_no;
3639
3640     for (pinno = pin_count; pinno < max_pin_count; ++pinno) {
3641         /* These pins are not specified but are not in the device.h
3642          * because the pin_table in the device.h only goes up to
3643          * the largest pin the user specifies.
3644          */
3645         soft_drive_low_count[palno][0] += 1;
3646     }
3647
3648     pin_spec_ptr = def_ptr->pin_table[0];
3649     for (pinno = 0; pinno < pin_count; ++pinno) {
3650
3651         unitno = ps_to_short_offset(pinno).unitno;
3652         palno = dab_ptr->unit_location(unitno).lane_no * MAX_SLOT_COUNT +
3653                 dab_ptr->unit_location(unitno).slot_no;
3654
3655         if (pin_spec_ptr->direction == NONE) {
3656             /* These pins will be soft driven LOW with minimum current */
3657             soft_drive_low_count[palno][0] += 1;
3658             ++pin_spec_ptr;
3659             continue;
3660         }
3661
3662         if ((pin_spec_ptr->direction == POWER) ||
3663             (pin_spec_ptr->direction == GROUND) ||
3664             (pin_spec_ptr->direction == NC)) {
3665             ++pin_spec_ptr;
3666             continue;
3667         }
3668
3669         /* SDLEN[3-0] */
3670         temp = pin_spec_ptr->s_drive_hi;
3671         if (temp & 0x8)
3672             soft_drive_high_count[palno][3] += 1;
3673         if (temp & 0x4)
3674             soft_drive_high_count[palno][2] += 1;
3675         if (temp & 0x2)
3676             soft_drive_high_count[palno][1] += 1;
3677         if (temp & 0x1)
3678             soft_drive_high_count[palno][0] += 1;
3679
3680         /* SDLEN[3-0]L */
3681         temp = pin_spec_ptr->s_drive_low;
3682         if (temp & 0x8)
3683             soft_drive_low_count[palno][3] += 1;
3684         if (temp & 0x4)
3685             soft_drive_low_count[palno][2] += 1;
3686         if (temp & 0x2)
3687             soft_drive_low_count[palno][1] += 1;
3688         if (temp & 0x1)
3689             soft_drive_low_count[palno][0] += 1;
3690
3691         ++pin_spec_ptr;
3692     }
3693
3694     for (palno = 0; palno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++palno) {
3695         /* Check the HIGH current limit */
3696         total_current =
3697             PEL_DRIVE_HIGH_3_CURRENT * soft_drive_high_count[palno][3] +
3698             PEL_DRIVE_HIGH_2_CURRENT * soft_drive_high_count[palno][2] +
3699             PEL_DRIVE_HIGH_1_CURRENT * soft_drive_high_count[palno][1] +
3700             PEL_DRIVE_HIGH_0_CURRENT * soft_drive_high_count[palno][0];
3701
3702         /* Adjust the total high current depending on how far vab is from vcc */
3703         total_current += (((def_ptr->vcc - def_ptr->vab) * 4) / 2500 + 1);
3704
3705         if (total_current > MAX_PEL_DRIVE_CURRENT) {
3706             current_limit_exceeded = TRUE;
3707             in_queue_message(ERROR_MSG, "Pin Electronics Module in lane: %c slot: %d high current limit exceeded",
3708                             'A' + palno / MAX_SLOT_COUNT,
3709                             palno % MAX_SLOT_COUNT);
3710         }
3711
3712         /* Check the LOW current limit */
3713         total_current =
3714             PEL_DRIVE_LOW_3_CURRENT * soft_drive_low_count[palno][3] +
3715             PEL_DRIVE_LOW_2_CURRENT * soft_drive_low_count[palno][2] +
3716             PEL_DRIVE_LOW_1_CURRENT * soft_drive_low_count[palno][1] +
3717             PEL_DRIVE_LOW_0_CURRENT * soft_drive_low_count[palno][0];
3718
3719     }
3720 }
3721
3722

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 6:14:53 pm | 32/182 |

```

SOURCE TEXT
LINE #
3721 if (total_current > MAX_PEL_DRIVE_CURRENT) {
3722     current_limit_exceeded = TRUE;
3723     la_queue_message(ERROR_MSG, "Pin Electronics Module is lane: tc slot: td low current limit exceeded",
3724         "A" * pelno / MAX_SLOT_COUNT,
3725         pelno * MAX_SLOT_COUNT);
3726 }
3727
3728 if (current_limit_exceeded == TRUE)
3729     return(FAILURE);
3730
3731 return(SUCCESS);
3732
3733 }
3734
3735 program_dac(def_ptr, dab_info_index, changed_dac)
3736 DEVICE_SPEC *def_ptr;
3737 char dab_info_index;
3738 u_char *changed_dac;
3739 {
3740     EXTRA_DEVICE_SPEC *extra_def_ptr;
3741     DAB_INFO *dab_ptr;
3742     u_long addr;
3743     u_char unitno;
3744     u_char total_unit;
3745     u_char lanes;
3746     u_char slots;
3747
3748     DPRINTF(("inside program_dac\n"));
3749
3750     *changed_dac = FALSE;
3751     extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
3752     dab_ptr = dab_list[dab_info_index];
3753
3754     total_unit = dab_ptr->unit_count;
3755     for (unitno = 0; unitno < total_unit; ++unitno) {
3756         lanes = dab_ptr->unit_locations[unitno].lane_no;
3757         slots = dab_ptr->unit_locations[unitno].slot_no;
3758
3759         addr = LANE_0_START_ADDR + lanes * LANE_ADDR_INC +
3760             LANE_PEL_0_START_OFFSET + slots * LANE_PEL_ADDR_INC;
3761
3762         if (dab_ptr->val_programmed != extra_def_ptr->val_dac_value) {
3763             DPRINTF(("program VSL\n"));
3764             write_loc_long((u_long *) (addr + PEL_VSL_DAC_WRITE_OFFSET),
3765                 (u_long) extra_def_ptr->val_dac_value);
3766             *changed_dac = TRUE;
3767         }
3768
3769         if (dab_ptr->vsh_programmed != extra_def_ptr->vsh_dac_value) {
3770             DPRINTF(("program VSH\n"));
3771             write_loc_long((u_long *) (addr + PEL_VSH_DAC_WRITE_OFFSET),
3772                 (u_long) extra_def_ptr->vsh_dac_value);
3773             *changed_dac = TRUE;
3774         }
3775
3776         if (dab_ptr->vth_programmed != extra_def_ptr->vth_dac_value) {
3777             DPRINTF(("program VTH\n"));
3778             write_loc_long((u_long *) (addr + PEL_VTH_DAC_WRITE_OFFSET),
3779                 (u_long) extra_def_ptr->vth_dac_value);
3780             *changed_dac = TRUE;
3781         }
3782
3783         if (dab_ptr->vwh_programmed != extra_def_ptr->vwh_dac_value) {
3784             DPRINTF(("program VWH\n"));
3785             write_loc_long((u_long *) (addr + PEL_VWH_DAC_WRITE_OFFSET),
3786                 (u_long) extra_def_ptr->vwh_dac_value);
3787             *changed_dac = TRUE;
3788         }
3789
3790         if (dab_ptr->vil_programmed != extra_def_ptr->vil_dac_value) {
3791             DPRINTF(("program VIL\n"));
3792             write_loc_long((u_long *) (addr + PEL_VIL_DAC_WRITE_OFFSET),
3793                 (u_long) extra_def_ptr->vil_dac_value);
3794             *changed_dac = TRUE;
3795         }
3796
3797         if (dab_ptr->vih_programmed != extra_def_ptr->vih_dac_value) {
3798             DPRINTF(("program VIH\n"));
3799             write_loc_long((u_long *) (addr + PEL_VIH_DAC_WRITE_OFFSET),
3800                 (u_long) extra_def_ptr->vih_dac_value);
3801             *changed_dac = TRUE;
3802         }
3803
3804         /* If there are no DAC change on the first unit of the DAB,
3805            * there won't be any on other units, so just break from the
3806            * for loop.
3807            */
3808         if (*changed_dac == FALSE)
3809             break;
3810     }
3811
3812 #ifdef MODELER
3813     if (*changed_dac == TRUE) {
3814         DPRINTF(("changed dac value\n"));
3815         setup_timer1(DAC_TIMER_VALUE);
3816         dab_ptr->val_programmed = extra_def_ptr->val_dac_value;
3817         dab_ptr->vsh_programmed = extra_def_ptr->vsh_dac_value;
3818         dab_ptr->vth_programmed = extra_def_ptr->vth_dac_value;
3819         dab_ptr->vwh_programmed = extra_def_ptr->vwh_dac_value;
3820         dab_ptr->vil_programmed = extra_def_ptr->vil_dac_value;
3821         dab_ptr->vih_programmed = extra_def_ptr->vih_dac_value;
3822     }
3823 #endif
3824
3825     return(SUCCESS);
3826 }
3827
3828 set_private_mode(dab_ptr, set_it)
3829 DAB_INFO *dab_ptr;
3830 u_char set_it;
3831 {
3832     /* If "set_it" == TRUE then set the PRIVATE bit on each PEL
3833        * otherwise reset the bit.
3834        */
3835     u_long addr;
3836     u_char total_unit;
3837     u_char value;
3838     u_char lanes;
3839 }
3840

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/util.c | DATE 5/23/89 | PAGE # 33/183 |
|--|--|---|-----------------|------------------|
| LINE # | | SOURCE TEXT | | |
| 3841 | | u_char slotno; | | |
| 3842 | | u_char unitno; | | |
| 3843 | | | | |
| 3844 | | total_unit = dab_ptr->unit_count; | | |
| 3845 | | for (unitno = 0; unitno < total_unit; ++unitno) { | | |
| 3846 | | lane0 = dab_ptr->unit_location[unitno].lane_no; | | |
| 3847 | | slotno = dab_ptr->unit_location[unitno].slot_no; | | |
| 3848 | | | | |
| 3849 | | addr = LANE_0_START_ADDR + lane0 * LANE_ADDR_INC + | | |
| 3850 | | LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC + | | |
| 3851 | | PEL_STATUS_CONTROL_OFFSET; | | |
| 3852 | | | | |
| 3853 | | value = read_loc_long((u_long *)addr) & 0xff; | | |
| 3854 | | if (set_it == TRUE) { | | |
| 3855 | | value = ~PEL_CS_PRIVATE_PUBLIC_MASK; | | |
| 3856 | | } | | |
| 3857 | | else { | | |
| 3858 | | value = ~PEL_CS_PRIVATE_PUBLIC_MASK; | | |
| 3859 | | } | | |
| 3860 | | write_loc_long((u_long *)addr, (u_long)value); | | |
| 3861 | | } | | |
| 3862 | | | | |
| 3863 | | | | |
| 3864 | | | | |
| 3865 | | turn_on_is_use(dab_ptr) | | |
| 3866 | | DAB_INFO *dab_ptr; | | |
| 3867 | | { | | |
| 3868 | | u_long addr; | | |
| 3869 | | u_char total_unit; | | |
| 3870 | | u_char value; | | |
| 3871 | | u_char lane0; | | |
| 3872 | | u_char slotno; | | |
| 3873 | | u_char unitno; | | |
| 3874 | | | | |
| 3875 | | total_unit = dab_ptr->unit_count; | | |
| 3876 | | for (unitno = 0; unitno < total_unit; ++unitno) { | | |
| 3877 | | lane0 = dab_ptr->unit_location[unitno].lane_no; | | |
| 3878 | | slotno = dab_ptr->unit_location[unitno].slot_no; | | |
| 3879 | | | | |
| 3880 | | addr = LANE_0_START_ADDR + lane0 * LANE_ADDR_INC + | | |
| 3881 | | LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC + | | |
| 3882 | | PEL_STATUS_CONTROL_OFFSET; | | |
| 3883 | | | | |
| 3884 | | value = read_loc_long((u_long *)addr) & 0xff ~PEL_CS_IN_USE_LED_MASK; | | |
| 3885 | | write_loc_long((u_long *)addr, (u_long)value); | | |
| 3886 | | } | | |
| 3887 | | | | |
| 3888 | | | | |
| 3889 | | | | |
| 3890 | | turn_off_is_use(dab_ptr) | | |
| 3891 | | DAB_INFO *dab_ptr; | | |
| 3892 | | { | | |
| 3893 | | u_long addr; | | |
| 3894 | | u_char total_unit; | | |
| 3895 | | u_char value; | | |
| 3896 | | u_char lane0; | | |
| 3897 | | u_char slotno; | | |
| 3898 | | u_char unitno; | | |
| 3899 | | | | |
| 3900 | | total_unit = dab_ptr->unit_count; | | |
| 3901 | | for (unitno = 0; unitno < total_unit; ++unitno) { | | |
| 3902 | | lane0 = dab_ptr->unit_location[unitno].lane_no; | | |
| 3903 | | slotno = dab_ptr->unit_location[unitno].slot_no; | | |
| 3904 | | | | |
| 3905 | | addr = LANE_0_START_ADDR + lane0 * LANE_ADDR_INC + | | |
| 3906 | | LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC + | | |
| 3907 | | PEL_STATUS_CONTROL_OFFSET; | | |
| 3908 | | | | |
| 3909 | | value = read_loc_long((u_long *)addr) & 0xff & ~PEL_CS_IN_USE_LED_MASK; | | |
| 3910 | | write_loc_long((u_long *)addr, (u_long)value); | | |
| 3911 | | } | | |
| 3912 | | | | |
| 3913 | | | | |
| 3914 | | | | |
| 3915 | | do_loop_ptr(instance) | | |
| 3916 | | INSTANCE_INFO *instance; | | |
| 3917 | | { | | |
| 3918 | | DEVICE_SPEC *def_ptr; | | |
| 3919 | | EXTRA_DEVICE_SPEC *extra_def_ptr; | | |
| 3920 | | u_long | | |
| 3921 | | inst_block_number[MAX_LANE_COUNT]; | | |
| 3922 | | u_long | | |
| 3923 | | seq_end_addr[MAX_LANE_COUNT]; | | |
| 3924 | | u_long | | |
| 3925 | | timeout; | | |
| 3926 | | u_char | | |
| 3927 | | changed_dac; | | |
| 3928 | | extern int | | |
| 3929 | | lm_intr_requested; /* global interrupt advisory flag */ | | |
| 3930 | | | | |
| 3931 | | lm_intr_requested = 0; | | |
| 3932 | | | | |
| 3933 | | /* The timeout value is set with the assumption that we are running at | | |
| 3934 | | 150MHz (8us period). | | |
| 3935 | | timeout = pattern_play_time * 1 second for feedback | | |
| 3936 | | = (pattern_count * 8 microsec + 125000 * 8 microsec | | |
| 3937 | | = (pattern_count * 125000) * 8 microsec | | |
| 3938 | | = (pattern_count * 125000) * 8 / 1000 millisecc | | |
| 3939 | | = (pattern_count * 125000) >> 7; | | |
| 3940 | | /* | | |
| 3941 | | timeout = (instance->pattern_count + PTHM_COUNT_FUDGE_FACTOR) >> 7; | | |
| 3942 | | | | |
| 3943 | | def_ptr = instance->definition; | | |
| 3944 | | extra_def_ptr = (EXTRA_DEVICE_SPEC *)extra_ptr->extra_data; | | |
| 3945 | | | | |
| 3946 | | if (program_dac(def_ptr, (char)instance->dab_info.index, | | |
| 3947 | | changed_dac) == FAILURE) { | | |
| 3948 | | return(FAILURE); | | |
| 3949 | | } | | |
| 3950 | | | | |
| 3951 | | #ifndef DBASE | | |
| 3952 | | if (start_tmng(def_ptr) == FAILURE) { | | |
| 3953 | | return(FAILURE); | | |
| 3954 | | } | | |
| 3955 | | #endif | | |
| 3956 | | if (set_edge_and_sample_setting(extra_def_ptr, | | |
| 3957 | | (u_long)RESOLUTION_05_NS, | | |
| 3958 | | (u_long)MAX_SAMPLE_RANGE) == FAILURE) | | |
| 3959 | | return(FAILURE); | | |
| 3960 | | set_seq_end_bit(instance, | | |
| 3961 | | instance->lane_addr, | | |
| 3962 | | FALSE; | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89
TIME 6:14:53 pm

PAGE #
34/184

```

LINE #          SOURCE TEXT
3961      seq_end_addr,
3962      inst_block_number);
3963
3964      for (i = 0; lm_inst_requested == 0; ++i) {
3965          if (play_ptrn_seq(instance, timeout, &changed_dac) == FAILURE) {
3966              remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
3967              if (i > 0)
3968                  lm_queue_message(ERROR_MSG, "failed in loop number: %d",
3969                                  i - lm_loop_patterns_count);
3970              return(FAILURE);
3971          }
3972      }
3973
3974      remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
3975      return(SUCCESS);
3976  }
3977
3978  count_avail_patterns(lanesum)
3979  u_char lanesum;
3980  {
3981      u_long total_block;
3982      u_long temp;
3983      u_char laneso;
3984
3985      if (lanesum == MAX_LANE_COUNT) {
3986          total_block = 0;
3987          for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso)
3988              DPRINTF(("avail on lane: ALL -> %d blocks\n", total_block));
3989          return(total_block);
3990      }
3991      else {
3992          total_block = 0;
3993          temp = free_block_list[lanesum];
3994          while (temp != NULL) {
3995              ++total_block;
3996              temp = read_loc_long((u_long *)temp);
3997          }
3998          DPRINTF(("avail on lane: %d -> %d blocks\n", lanesum, total_block));
3999          return(total_block);
4000      }
4001  }
4002
4003  lm_free(ptr)
4004  char *ptr;
4005  {
4006      DPRINTF(("lm_free: %08X\n", ptr));
4007      free(ptr);
4008  }
4009
4010  char *
4011  lm_calloc(nelem, elsize)
4012  unsigned nelem, elsize;
4013  {
4014      char *ptr;
4015
4016      DPRINTF(("lm_calloc: %d elements of %d bytes -> ", nelem, elsize));
4017      ptr = calloc(nelem, elsize);
4018      DPRINTF((" %08X\n", ptr));
4019      return(ptr);
4020  }
4021
4022  char *
4023  lm_malloc(size)
4024  unsigned size;
4025  {
4026      char *ptr;
4027
4028      DPRINTF(("lm_malloc: %d bytes -> ", size));
4029      ptr = malloc(size);
4030      DPRINTF((" %08X\n", ptr));
4031      return(ptr);
4032  }
4033
4034  char *
4035  lm_realloc(ptr, size)
4036  char *ptr;
4037  unsigned size;
4038  {
4039      char *ptr2;
4040
4041      DPRINTF(("lm_realloc: ptr = %08X size = %08X -> ", ptr, size));
4042      ptr2 = realloc(ptr, size);
4043      DPRINTF((" %08X\n", ptr2));
4044      return(ptr2);
4045  }
4046
4047  get_fatal_hardware_message()
4048  {
4049      u_char laneso;
4050
4051      if (modeler_error.tmg_error == TRUE)
4052          lm_queue_message(ERROR_MSG, "fatal Timing Generator error");
4053
4054      if (modeler_error.unknown_source_of_interrupt == TRUE) {
4055          for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {
4056              if (! (modeler_error.lane_errors & (1 << laneso)))
4057                  continue;
4058              lm_queue_message(ERROR_MSG, "unknown source of interrupt on lane: %c",
4059                              'A' + laneso);
4060          }
4061      }
4062
4063      if (modeler_error.lane_errors) {
4064          for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {
4065              if (! (modeler_error.lane_errors & (1 << laneso)))
4066                  continue;
4067              if (modeler_error.pac_error[laneso].pac_refresh_error == TRUE)
4068                  lm_queue_message(ERROR_MSG, "fatal Pattern Controller refresh error in lane: %c",
4069                              'A' + laneso);
4070              if (modeler_error.pac_error[laneso].pac_request_error == TRUE)
4071                  lm_queue_message(ERROR_MSG, "fatal Pattern Controller request error in lane: %c",
4072                              'A' + laneso);
4073              if (modeler_error.pac_error[laneso].pac_pattern_error == TRUE)
4074                  lm_queue_message(ERROR_MSG, "fatal Pattern Controller pattern error in lane: %c",
4075                              'A' + laneso);
4076              if (modeler_error.pac_error[laneso].
4077                  pac_control_word_parity_error == TRUE)

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89
TIME 6:14:53 pm

PAGE #
35/185

```

LINE #          SOURCE TEXT
4081      lm_queue_message(ERROR_MSG, "Fatal Pattern Controller CTL word parity error in lane: %c",
4082      'A' + lane_no);
4083      if (modeler_error.pac_error[lane_no].
4084      pac_high_word_parity_error == TRUE)
4085      lm_queue_message(ERROR_MSG, "Fatal Pattern Controller high word parity error in lane: %c",
4086      'A' + lane_no);
4087      if (modeler_error.pac_error[lane_no].
4088      pac_low_word_parity_error == TRUE)
4089      lm_queue_message(ERROR_MSG, "Fatal Pattern Controller low word parity error in lane: %c",
4090      'A' + lane_no);
4091      }
4092      }
4093      }
4094      }
4095      check_pel_errors(instance);
4096      INSTANCE_INFO "instance:
4097      {
4098      DAB_INFO      *dab_ptr,
4099      u_long      pel_error_list;
4100      u_long      mask;
4101      u_char      cur_pelno;
4102      u_char      pelno;
4103      u_char      magicno;
4104      u_char      unitno;
4105      u_char      magic_mask;
4106      u_char      found_error;
4107      u_char      error_on_this_instance = FALSE;
4108      }
4109      dab_ptr = dab_list(instance->dab_info_index);
4110      }
4111      pel_error_list = modeler_error.pel_error_list;
4112      modeler_error.pel_error_list = 0;
4113      for (pelno = 0; pelno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++pelno) {
4114      if (pel_error_list == 0)
4115      break;
4116      }
4117      found_error = FALSE;
4118      }
4119      mask = 1 << pelno;
4120      if (! (pel_error_list & mask))
4121      continue;
4122      }
4123      if (system_config->lane[pelno / MAX_SLOT_COUNT]->
4124      pel[pelno % MAX_SLOT_COUNT] == NULL)
4125      continue;
4126      }
4127      pel_error_list ^= mask;
4128      }
4129      /* Check PLAY error */
4130      if (modeler_error.pel_error[pelno].play_error == TRUE) {
4131      lm_queue_message(ERROR_MSG, "play error in lane: %c slot: %d",
4132      'A' + pelno / MAX_SLOT_COUNT,
4133      pelno % MAX_SLOT_COUNT);
4134      found_error = TRUE;
4135      modeler_error.pel_error[pelno].play_error = FALSE;
4136      }
4137      }
4138      /* Check MAGIC PARITY error */
4139      if (modeler_error.pel_error[pelno].any_magic_parity == TRUE) {
4140      found_error = TRUE;
4141      modeler_error.pel_error[pelno].any_magic_parity = FALSE;
4142      }
4143      for (magicno = 0; magicno < MAX_MAGIC_COUNT; ++magicno) {
4144      magic_mask = 1 << magicno;
4145      if ((modeler_error.pel_error[pelno].
4146      magic_parity_out & magic_mask) != 0) {
4147      lm_queue_message(ERROR_MSG, "Pin Electronics Module parity error in lane: %c slot: %d channel: %d",
4148      'A' + pelno / MAX_SLOT_COUNT,
4149      pelno % MAX_SLOT_COUNT,
4150      magicno);
4151      modeler_error.pel_error[pelno].magic_parity_out ^= magic_mask;
4152      }
4153      }
4154      }
4155      }
4156      /* Check MAGIC SHORT error */
4157      if (modeler_error.pel_error[pelno].any_magic_short == TRUE) {
4158      /* Find the unitno of this pel */
4159      for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
4160      cur_pelno =
4161      dab_ptr->unit_location[unitno].lane_no * 8 +
4162      dab_ptr->unit_location[unitno].slot_no;
4163      if (cur_pelno == pelno) {
4164      break;
4165      }
4166      }
4167      }
4168      }
4169      }
4170      if (unitno == dab_ptr->unit_count) {
4171      mark_instance_shorted_pin(pelno);
4172      init_dab(pelno / MAX_SLOT_COUNT, pelno % MAX_SLOT_COUNT);
4173      continue;
4174      }
4175      }
4176      found_error = TRUE;
4177      }
4178      report_shorted_pin(instance);
4179      }
4180      }
4181      if (found_error == TRUE) {
4182      error_on_this_instance = TRUE;
4183      init_dab(pelno / MAX_SLOT_COUNT, pelno % MAX_SLOT_COUNT);
4184      }
4185      }
4186      }
4187      if (error_on_this_instance == TRUE)
4188      return(FAILURE);
4189      }
4190      return(SUCCESS);
4191      }
4192      }
4193      mark_instance_shorted_pin(pelno)
4194      u_char pelno;
4195      {
4196      USER_INFO      *user;
4197      INSTANCE_INFO  *instance;
4198      DAB_INFO      *dab_ptr;
4199      u_short      inst_id;
4200      u_char      username;

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89 PAGE #
TIME 6:14:53 pm 36/186

```

LINE # SOURCE TEXT
4201 u_char          unitno;
4202 u_char          cur_pelso;
4203
4204 for (userno = 0; userno < MAX_USER_COUNT; ++userno) {
4205     user = user_info_array[userno];
4206
4207     if (user->active == FALSE)
4208         continue;
4209
4210     for (inst_id = 0; inst_id < user->inst_table_size; ++inst_id) {
4211
4212         instance = user->instance[inst_id];
4213
4214         if (IS_MAGIC_INSTANCE(user, instance))
4215             continue;
4216
4217         dab_ptr = dab_list[instance->dab_info_index];
4218
4219         for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
4220             cur_pelso =
4221                 dab_ptr->unit_location[unitno].lane_no * 8 +
4222                 dab_ptr->unit_location[unitno].slot_no;
4223             if (cur_pelso == pelso) {
4224                 /* This instance is using the pelso */
4225                 instance->had_shorted_pin = TRUE;
4226                 return;
4227             }
4228         }
4229     }
4230 }
4231
4232 report_shorted_pin(instance)
4233 INSTANCE_INFO *instance;
4234 {
4235     DEVICE_SPEC *definition;
4236     DAB_INFO *dab_ptr;
4237     magic_short;
4238     u_short      pin_number;
4239     u_short      bit_mask;
4240     u_char        unitno;
4241     u_char        pelso;
4242     u_char        magicno;
4243     u_char        bitno;
4244     u_char        unit_pin_number;
4245     u_char        direction;
4246
4247     dab_ptr = dab_list[instance->dab_info_index];
4248     definition = instance->definition;
4249
4250     for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
4251         pelso =
4252             dab_ptr->unit_location[unitno].lane_no * 8 +
4253             dab_ptr->unit_location[unitno].slot_no;
4254
4255         if (modeler_error.pel_error[pelso].any_magic_short == TRUE) {
4256             modeler_error.pel_error[pelso].any_magic_short = FALSE;
4257
4258             for (magicno = 0; magicno < MAX_MAGIC_COUNT; ++magicno) {
4259                 magic_short = modeler_error.pel_error[pelso].magic_short[magicno];
4260                 modeler_error.pel_error[pelso].magic_short[magicno] = 0;
4261
4262                 if (magic_short != 0) {
4263                     for (bitno = 0; bitno < 16; ++bitno) {
4264
4265                         if (magic_short == 0)
4266                             break;
4267
4268                         bit_mask = 1 << bitno;
4269
4270                         if (!(magic_short & bit_mask))
4271                             continue;
4272
4273                         magic_short ^= bit_mask;
4274
4275                         unit_pin_number = magicno * 16 + bitno;
4276
4277                         pin_number = unitno * 80 + unit_pin_number;
4278                         direction = definition->pin_table[pin_number].direction;
4279                         if ((direction == NONE) ||
4280                             (direction == POWER) ||
4281                             (direction == GROUND) ||
4282                             (direction == NC))
4283                             continue;
4284                         in_queue_message(ERROR_MSG, "internal error: shorted NONE/POWER/GROUND/NC pin in lane: %c slot: %d unit PN: %d",
4285                                         'A' + pelso / MAX_SLOT_COUNT,
4286                                         pelso % MAX_SLOT_COUNT,
4287                                         unit_pin_number);
4288                     }
4289                     in_queue_message(ERROR_MSG, "pin: %s of instance: %s is shorted",
4290                                     definition->pin_table[pin_number].pin_name,
4291                                     instance->device_info_string);
4292                 }
4293             }
4294         }
4295     }
4296 }
4297
4298 init_err()
4299 {
4300     u_long junk;
4301     u_char lane0;
4302     u_char pelso;
4303     u_char magicno;
4304
4305     if (read_tag(&junk) == SUCCESS) {
4306         tagptr->lane_intr_enable = 1;
4307         tagptr->tag_intr_enable = 1;
4308         tagptr->tag_intr_clear1 = 1;
4309     }
4310
4311     modeler_error.pel_error_list = 0;
4312     modeler_error.error = FALSE;
4313     modeler_error.lane_errors = 0;
4314     modeler_error.pac_lane_errors = 0;
4315     modeler_error.dab_change = FALSE;
4316     modeler_error.tag_error = FALSE;
4317
4318     for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
4319         modeler_error.pac_error[lane0].pac_refresh_error = FALSE;
4320     }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89
TIME 6:14:53 pm

PAGE #
37/187

```

LINE # SOURCE TEXT
4321 modeler_error.pac_error(lasemo).pac_request_error = FALSE;
4322 modeler_error.pac_error(lasemo).pac_patterns_error = FALSE;
4323 modeler_error.pac_error(lasemo).pac_control_word_parity_error = FALSE;
4324 modeler_error.pac_error(lasemo).pac_high_word_parity_error = FALSE;
4325 modeler_error.pac_error(lasemo).low_word_parity_error = FALSE;
4326 modeler_error.pac_error(lasemo).branch_address = 0;
4327 modeler_error.pac_error(lasemo).pac_block_offset = 0;
4328 modeler_error.pac_error(lasemo).pac_parity_error_address = 0;
4329 }
4330
4331 for (pelno = 0; pelno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++pelno) {
4332     modeler_error.pel_error(pelno).dab_inserted = FALSE;
4333     modeler_error.pel_error(pelno).dab_removed = FALSE;
4334     modeler_error.pel_error(pelno).play_error = FALSE;
4335     modeler_error.pel_error(pelno).asy_magic_short = FALSE;
4336     modeler_error.pel_error(pelno).asy_magic_parity = FALSE;
4337     modeler_error.pel_error(pelno).magic_parity_out = 0;
4338 }
4339
4340 for (magicno = 0; magicno < 5; ++magicno) {
4341     modeler_error.pel_error(pelno).magic_short[magicno] = 0;
4342 }
4343 }
4344
4345 clear_non_fatal_mod_err()
4346 {
4347     u_char pelno;
4348     u_char magicno;
4349
4350     modeler_error.pel_error_list = 0;
4351     for (pelno = 0; pelno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++pelno) {
4352         modeler_error.pel_error(pelno).dab_inserted = FALSE;
4353         modeler_error.pel_error(pelno).dab_removed = FALSE;
4354         modeler_error.pel_error(pelno).play_error = FALSE;
4355         modeler_error.pel_error(pelno).asy_magic_short = FALSE;
4356         modeler_error.pel_error(pelno).asy_magic_parity = FALSE;
4357         modeler_error.pel_error(pelno).magic_parity_out = 0;
4358     }
4359     for (magicno = 0; magicno < 5; ++magicno) {
4360         modeler_error.pel_error(pelno).magic_short[magicno] = 0;
4361     }
4362 }
4363
4364
4365 lm_config_error(error_type, lasemo, slotno)
4366 u_char error_type;
4367 u_char lasemo;
4368 u_char slotno;
4369 {
4370     switch (error_type) {
4371     case CERR_TWG_CAL:
4372         config_error.twg_cal = 1;
4373         fatal_configuration_error_encountered = TRUE;
4374         break;
4375     case CERR_NO_TWG:
4376         config_error.no_twg = 1;
4377         fatal_configuration_error_encountered = TRUE;
4378         break;
4379     case CERR_NO_PAM_FOR_PAC:
4380         config_error.no_pam_for_pac |= 1 << (lasemo * MAX_PAM_COUNT + slotno);
4381         fatal_configuration_error_encountered = TRUE;
4382         break;
4383     case CERR_PAM_STRAPPED_WRONG:
4384         config_error.pam_strapped_wrong |=
4385             1 << (lasemo * MAX_PAM_COUNT + slotno);
4386         fatal_configuration_error_encountered = TRUE;
4387         break;
4388     case CERR_PAM_STACKED_WRONG:
4389         config_error.pam_stacked_wrong |=
4390             1 << (lasemo * MAX_PAM_COUNT + slotno);
4391         fatal_configuration_error_encountered = TRUE;
4392         break;
4393     case CERR_DUPLICATE_SEGMENT:
4394         config_error.duplicate_segment |=
4395             1 << (lasemo * MAX_SLOT_COUNT + slotno);
4396         non_fatal_configuration_error_encountered = TRUE;
4397         break;
4398     case CERR_MISSING_SEGMENT:
4399         config_error.missing_segment |=
4400             1 << (lasemo * MAX_SLOT_COUNT + slotno);
4401         non_fatal_configuration_error_encountered = TRUE;
4402         break;
4403     case CERR_NO_PEL_FOR_DAB:
4404         config_error.no_pel_for_dab |=
4405             1 << (lasemo * MAX_SLOT_COUNT + slotno);
4406         non_fatal_configuration_error_encountered = TRUE;
4407         break;
4408     case CERR_NO_PAM_FOR_DAB:
4409         config_error.no_pam_for_dab |=
4410             1 << (lasemo * MAX_SLOT_COUNT + slotno);
4411         non_fatal_configuration_error_encountered = TRUE;
4412         break;
4413     case CERR_ILLEGAL_DUPLICATE_DEVICE:
4414         config_error.illegal_duplicate_device |=
4415             1 << (lasemo * MAX_SLOT_COUNT + slotno);
4416         non_fatal_configuration_error_encountered = TRUE;
4417         break;
4418     case CERR_DEVICE_TOO_LARGE:
4419         config_error.device_too_large |=
4420             1 << (lasemo * MAX_SLOT_COUNT + slotno);
4421         non_fatal_configuration_error_encountered = TRUE;
4422         break;
4423     case CERR_ILLEGAL_PEL_STACKING:
4424         config_error.illegal_pel_stacking |=
4425             1 << (lasemo * MAX_SLOT_COUNT + slotno);
4426         non_fatal_configuration_error_encountered = TRUE;
4427         break;
4428     default:
4429         DPRINTF(("illegal error_type in config_error\n"));
4430         break;
4431     }
4432 }
4433
4434 init_config_error()
4435 {
4436     u_long error;
4437
4438     config_error.do_twg_calibration = 0;
4439
4440 #ifdef MODELER

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/util.c | DATE 5/23/89 | PAGE # 38/188 |
|--|--|--|-----------------|------------------|
| LINE # | | SOURCE TEXT | | |
| 4441 | | (void)lm_svaram_access((char *)&config_error, CONFIG_ERROR, (u_long)sizeof(CONFIGURATION_ERRORS), MEMORY_READ, &error); | | |
| 4442 | | | | |
| 4443 | | #endif | | |
| 4444 | | | | |
| 4445 | | config_error.tmg_cal = 0; | | |
| 4446 | | config_error.no_tmg = 0; | | |
| 4447 | | config_error.no_pam_for_pac = 0; | | |
| 4448 | | config_error.pam_strapped_wrong = 0; | | |
| 4449 | | config_error.pam_stacked_wrong = 0; | | |
| 4450 | | config_error.duplicate_segment = 0; | | |
| 4451 | | config_error.missing_segment = 0; | | |
| 4452 | | config_error.no_pai_for_dab = 0; | | |
| 4453 | | config_error.no_pam_for_dab = 0; | | |
| 4454 | | config_errorillegal_duplicate_device = 0; | | |
| 4455 | | config_error.device_too_large = 0; | | |
| 4456 | | config_errorillegal_pai_stacking = 0; | | |
| 4457 | | | | |
| 4458 | | } | | |
| 4459 | | write_config_error(); | | |
| 4460 | | { | | |
| 4461 | | #ifdef MODELER | | |
| 4462 | | u_long error; | | |
| 4463 | | DPRINTF(("write config_error: CONFIG_ERROR: %08x, size: %d\n", | | |
| 4464 | | CONFIG_ERROR, sizeof(CONFIGURATION_ERRORS))); | | |
| 4465 | | | | |
| 4466 | | DPRINTF(("tmg_cal = %d\n", | | |
| 4467 | | config_error.tmg_cal)); | | |
| 4468 | | DPRINTF(("no_tmg = %d\n", | | |
| 4469 | | config_error.no_tmg)); | | |
| 4470 | | DPRINTF(("no_pam_for_pac = %08x\n", | | |
| 4471 | | config_error.no_pam_for_pac)); | | |
| 4472 | | DPRINTF(("pam_strapped_wrong = %08x\n", | | |
| 4473 | | config_error.pam_strapped_wrong)); | | |
| 4474 | | DPRINTF(("pam_stacked_wrong = %08x\n", | | |
| 4475 | | config_error.pam_stacked_wrong)); | | |
| 4476 | | DPRINTF(("duplicate_segment = %08x\n", | | |
| 4477 | | config_error.duplicate_segment)); | | |
| 4478 | | DPRINTF(("missing_segment = %08x\n", | | |
| 4479 | | config_error.missing_segment)); | | |
| 4480 | | DPRINTF(("no_pai_for_dab = %08x\n", | | |
| 4481 | | config_error.no_pai_for_dab)); | | |
| 4482 | | DPRINTF(("no_pam_for_dab = %08x\n", | | |
| 4483 | | config_error.no_pam_for_dab)); | | |
| 4484 | | DPRINTF(("illegal_duplicate_device = %08x\n", | | |
| 4485 | | config_errorillegal_duplicate_device)); | | |
| 4486 | | DPRINTF(("device_too_large = %08x\n", | | |
| 4487 | | config_error.device_too_large)); | | |
| 4488 | | DPRINTF(("illegal_pai_stacking = %08x\n", | | |
| 4489 | | config_errorillegal_pai_stacking)); | | |
| 4490 | | | | |
| 4491 | | (void)lm_svaram_access((char *)&config_error, CONFIG_ERROR, (u_long)sizeof(CONFIGURATION_ERRORS), MEMORY_WRITE, &error); | | |
| 4492 | | | | |
| 4493 | | #endif | | |
| 4494 | | | | |
| 4495 | | } | | |
| 4496 | | write_user_stat(pattern_count, pattern_count1, longest_pattern_seq, def_count, inst_count, fault_count); | | |
| 4497 | | u_long pattern_count; | | |
| 4498 | | u_long pattern_count1; | | |
| 4499 | | u_long longest_pattern_seq; | | |
| 4500 | | u_short def_count; | | |
| 4501 | | u_short inst_count; | | |
| 4502 | | u_short fault_count; | | |
| 4503 | | #ifdef MODELER | | |
| 4504 | | u_long error; | | |
| 4505 | | #endif | | |
| 4506 | | RUNTIME_STAT_STRUCT svaram_value; | | |
| 4507 | | #ifdef MODELER | | |
| 4508 | | (void)lm_svaram_access((char *)&svaram_value, RUNTIME_STAT, (u_long)sizeof(RUNTIME_STAT_STRUCT), MEMORY_READ, &error); | | |
| 4509 | | | | |
| 4510 | | #endif | | |
| 4511 | | add_64(&svaram_value.pattern_count1, &svaram_value.pattern_count1, &pattern_count1, pattern_count1); | | |
| 4512 | | | | |
| 4513 | | if (&svaram_value.longest_pattern_seq < longest_pattern_seq) | | |
| 4514 | | svaram_value.longest_pattern_seq = longest_pattern_seq; | | |
| 4515 | | | | |
| 4516 | | svaram_value.definition_count += def_count; | | |
| 4517 | | svaram_value.instance_count += inst_count; | | |
| 4518 | | svaram_value.fault_count += fault_count; | | |
| 4519 | | | | |
| 4520 | | #ifdef MODELER | | |
| 4521 | | (void)lm_svaram_access((char *)&svaram_value, RUNTIME_STAT, (u_long)sizeof(RUNTIME_STAT_STRUCT), MEMORY_WRITE, &error); | | |
| 4522 | | | | |
| 4523 | | #endif | | |
| 4524 | | add_64(op1h, op1l, op2h, op2l) | | |
| 4525 | | u_long op1h; | | |
| 4526 | | u_long op1l; | | |
| 4527 | | u_long op2h; | | |
| 4528 | | u_long op2l; | | |
| 4529 | | { | | |
| 4530 | | /* 64 bit add: *op1 <- *op1 + *op2 */ | | |
| 4531 | | u_long th; | | |
| 4532 | | u_long tl; | | |
| 4533 | | th = *op1 + *op2h; | | |
| 4534 | | tl = *op1 + *op2l; | | |
| 4535 | | | | |
| 4536 | | if (((*op1 & 0x80000000) & (*op2h & 0x80000000)) | | |
| 4537 | | (((*op1 & 0x80000000) & (*op2l & 0x80000000)) & (*tl & 0x80000000))) | | |
| 4538 | | ++th; | | |
| 4539 | | *op1h = th; | | |
| 4540 | | *op1l = tl; | | |
| 4541 | | } | | |
| 4542 | | | | |
| 4543 | | | | |
| 4544 | | | | |
| 4545 | | | | |
| 4546 | | | | |
| 4547 | | | | |
| 4548 | | | | |
| 4549 | | | | |
| 4550 | | | | |
| 4551 | | | | |
| 4552 | | | | |
| 4553 | | | | |
| 4554 | | | | |
| 4555 | | | | |
| 4556 | | | | |
| 4557 | | | | |
| 4558 | | | | |
| 4559 | | | | |
| 4560 | | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
lm1000/util.c

DATE 5/23/89
TIME 6:14:53 pm

PAGE #
39/189

```

LINE # SOURCE TEXT
4561 update_svarm_dab_insertion(loc_dab_list)
4562 DAB_INFO *loc_dab_list;
4563 {
4564     DAB_INFO *dab_ptr;
4565     #ifdef MODELER
4566     .b esprun;
4567     u_long svarm_dab_insertion;
4568     u_long error;
4569     #endif
4570     u_char dab_insertion = 0;
4571     u_char dabno;
4572     char segno;
4573
4574     for (dabno = 0; dabno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++dabno) {
4575         if (loc_dab_list[dabno] == NULL)
4576             continue;
4577         dab_ptr = loc_dab_list[dabno];
4578
4579         for (segno = 0; segno <= MAX_SEGMENT_PER_DEVICE; ++segno) {
4580             if (dab_ptr->segment[segno] == NULL)
4581                 continue;
4582
4583             if (dab_active(dab_ptr->segment[segno]->lane_no,
4584                 dab_ptr->segment[segno]->slot_no) == FALSE) {
4585                 ++dab_insertion;
4586             }
4587         }
4588     }
4589     #ifdef MODELER
4590     (void)lm_write_esprun_in_count(
4591         dab_ptr->segment[segno]->lane_no * MAX_SLOT_COUNT +
4592         dab_ptr->segment[segno]->slot_no,
4593         (u_short)dab_ptr->segment[segno]->insertion_count + 1);
4594     /* ??? */
4595     (void)lm_read_esprun(
4596         dab_ptr->segment[segno]->lane_no * MAX_SLOT_COUNT +
4597         dab_ptr->segment[segno]->slot_no,
4598         &dab_esprun);
4599
4600     if (dab_esprun.insertion_count !=
4601         dab_ptr->segment[segno]->insertion_count + 1) {
4602         DPRINTF(("DAB EEPROM write insertion_count failed\n"));
4603         DPRINTF(("wrote: %d read: %d\n",
4604             dab_ptr->segment[segno]->insertion_count + 1,
4605             dab_esprun.insertion_count));
4606     }
4607     #endif
4608 }
4609
4610 #endif
4611
4612 #ifdef MODELER
4613 (void)lm_svarm_access((char *)&svarm_dab_insertion, RUNTIME_STAT,
4614     (u_long)sizeof(u_long), MEMORY_READ,
4615     &error);
4616 svarm_dab_insertion += dab_insertion;
4617
4618 (void)lm_svarm_access((char *)&svarm_dab_insertion, RUNTIME_STAT,
4619     (u_long)sizeof(u_long), MEMORY_WRITE,
4620     &error);
4621 #endif
4622
4623 dab_active(laneno, slotno)
4624 u_char laneno;
4625 u_char slotno;
4626 {
4627     u_long addr;
4628     u_short value;
4629
4630     addr = LANE_0_START_ADDR + laneno * LANE_ADDR_INC +
4631         LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC +
4632         PEL_STATUS_CONTROL_OFFSET;
4633
4634     value = (u_short)read_loc_long((u_long *)&addr);
4635
4636     if (((value & PEL_CS_ACTIVE_MASK) == 0) ||
4637         ((value & PEL_CS_INITIALIZE_MASK) == 0) ||
4638         ((value & PEL_CS_RESET_MASK) == 0))
4639         return(FALSE);
4640
4641     return(TRUE);
4642 }
4643
4644 fatal_alloc_error()
4645 {
4646     /* ??? */
4647 }
4648
4649 add_inconsistent_pins(instance, ident_inconsistent_pins)
4650 INSTANCE_INFO *instance;
4651 PTRN_BITS_LONGWORD *ident_inconsistent_pins;
4652 {
4653     DAB_INFO *dab_ptr;
4654     u_long *unknown_ptr;
4655     u_char total_word;
4656     u_char wordno;
4657
4658     dab_ptr = dab_list[instance->dab_info_index];
4659     total_word = dab_ptr->unit_count *
4660         sizeof(PTRN_BITS_LONGWORD) / sizeof(u_long);
4661
4662     unknown_ptr = (u_long *)instance->last_sample_value.unknown;
4663     for (wordno = 0; wordno < total_word; ++wordno) {
4664         unknown_ptr[wordno] |= ((u_long *)ident_inconsistent_pins)[wordno];
4665     }
4666 }
4667
4668 epoch_info(addr, value)
4669 u_short addr;
4670 u_long *value;
4671 {
4672     #ifdef MODELER
4673     u_long error;
4674     RUNTIME_STAT_STRUCT rstat;
4675     #endif

```


| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/util.c | DATE 5/23/89 | PAGE # 40/190 |
|--|--|--|-----------------|------------------|
| LINE # | | SOURCE TEXT | | |
| 4681 | | (void)lm_aware_access((char *)rstat, RUNTIME_STAT, | | |
| 4682 | | (u_long)sizeof(RUNTIME_STAT_STRUCT), MEMORY_READ, | | |
| 4683 | | &error); | | |
| 4684 | | | | |
| 4685 | | | | |
| 4686 | | switch (attr) { | | |
| 4687 | | case LM_EPOCH_ADAPTER_INSERTIONS: | | |
| 4688 | | *value = rstat.dab_insertion_count; | | |
| 4689 | | break; | | |
| 4690 | | case LM_EPOCH_LONGEST_SEQUENCE: | | |
| 4691 | | *value = rstat.longest_pattern_seq; | | |
| 4692 | | break; | | |
| 4693 | | case LM_EPOCH_PATTERNS_ADDED_HIGH: | | |
| 4694 | | *value = rstat.pattern_count_high; | | |
| 4695 | | break; | | |
| 4696 | | case LM_EPOCH_PATTERNS_ADDED_LOW: | | |
| 4697 | | *value = rstat.pattern_count_low; | | |
| 4698 | | break; | | |
| 4699 | | case LM_EPOCH_DEFINITIONS: | | |
| 4700 | | *value = rstat.definition_count; | | |
| 4701 | | break; | | |
| 4702 | | case LM_EPOCH_INSTANCES: | | |
| 4703 | | *value = rstat.instance_count; | | |
| 4704 | | break; | | |
| 4705 | | case LM_EPOCH_FAULTS: | | |
| 4706 | | *value = rstat.fault_count; | | |
| 4707 | | break; | | |
| 4708 | | default: | | |
| 4709 | | return(FAILURE); | | |
| 4710 | | } | | |
| 4711 | | if (attr == LM_EPOCH_ADAPTER_INSERTIONS) | | |
| 4712 | | { | | |
| 4713 | | switch (attr) { | | |
| 4714 | | case LM_EPOCH_ADAPTER_INSERTIONS: | | |
| 4715 | | *value = rstat.dab_insertion_count; | | |
| 4716 | | case LM_EPOCH_LONGEST_SEQUENCE: | | |
| 4717 | | *value = rstat.longest_pattern_seq; | | |
| 4718 | | case LM_EPOCH_PATTERNS_ADDED_HIGH: | | |
| 4719 | | *value = rstat.pattern_count_high; | | |
| 4720 | | case LM_EPOCH_PATTERNS_ADDED_LOW: | | |
| 4721 | | *value = rstat.pattern_count_low; | | |
| 4722 | | case LM_EPOCH_DEFINITIONS: | | |
| 4723 | | *value = rstat.definition_count; | | |
| 4724 | | case LM_EPOCH_INSTANCES: | | |
| 4725 | | *value = rstat.instance_count; | | |
| 4726 | | case LM_EPOCH_FAULTS: | | |
| 4727 | | *value = rstat.fault_count; | | |
| 4728 | | default: | | |
| 4729 | | return(FAILURE); | | |
| 4730 | | } | | |
| 4731 | | } | | |
| 4732 | | return(SUCCESS); | | |
| 4733 | | } | | |
| 4734 | | adjust_delay(user, def_ptr) | | |
| 4735 | | USER_INFO *user; | | |
| 4736 | | DEVICE_SPEC *def_ptr; | | |
| 4737 | | { | | |
| 4738 | | u_short i; | | |
| 4739 | | u_short mtyp_count; | | |
| 4740 | | /* Adjust the default delay if specified, otherwise set it to 1.0 */ | | |
| 4741 | | if (def_ptr->use_default == TRUE) { | | |
| 4742 | | ADJUST_DELAY(def_ptr->default_delay.minimum, | | |
| 4743 | | user->time_scale, | | |
| 4744 | | user->half_time_scale, | | |
| 4745 | | user->divide_delay); | | |
| 4746 | | ADJUST_DELAY(def_ptr->default_delay.typical, | | |
| 4747 | | user->time_scale, | | |
| 4748 | | user->half_time_scale, | | |
| 4749 | | user->divide_delay); | | |
| 4750 | | ADJUST_DELAY(def_ptr->default_delay.maximum, | | |
| 4751 | | user->time_scale, | | |
| 4752 | | user->half_time_scale, | | |
| 4753 | | user->divide_delay); | | |
| 4754 | | } | | |
| 4755 | | else { | | |
| 4756 | | def_ptr->default_delay.minimum = 1; | | |
| 4757 | | def_ptr->default_delay.typical = 1; | | |
| 4758 | | def_ptr->default_delay.maximum = 1; | | |
| 4759 | | } | | |
| 4760 | | mtyp_count = def_ptr->mtyp_count; | | |
| 4761 | | for (i = 0; i < mtyp_count; ++i) { | | |
| 4762 | | if ((long)def_ptr->mtyp_table[i].minimum != -1) | | |
| 4763 | | ADJUST_DELAY(def_ptr->mtyp_table[i].minimum, | | |
| 4764 | | user->time_scale, | | |
| 4765 | | user->half_time_scale, | | |
| 4766 | | user->divide_delay); | | |
| 4767 | | if ((long)def_ptr->mtyp_table[i].typical != -1) | | |
| 4768 | | ADJUST_DELAY(def_ptr->mtyp_table[i].typical, | | |
| 4769 | | user->time_scale, | | |
| 4770 | | user->half_time_scale, | | |
| 4771 | | user->divide_delay); | | |
| 4772 | | if ((long)def_ptr->mtyp_table[i].maximum != -1) | | |
| 4773 | | ADJUST_DELAY(def_ptr->mtyp_table[i].maximum, | | |
| 4774 | | user->time_scale, | | |
| 4775 | | user->half_time_scale, | | |
| 4776 | | user->divide_delay); | | |
| 4777 | | } | | |
| 4778 | | unadjust_delay(user, def_ptr) | | |
| 4779 | | USER_INFO *user; | | |
| 4780 | | DEVICE_SPEC *def_ptr; | | |
| 4781 | | { | | |
| 4782 | | u_short i; | | |
| 4783 | | u_short mtyp_count; | | |
| 4784 | | /* Unadjust the default delay if specified */ | | |
| 4785 | | if (def_ptr->use_default == TRUE) { | | |
| 4786 | | UNADJUST_DELAY(def_ptr->default_delay.minimum, | | |
| 4787 | | user->time_scale, | | |
| 4788 | | user->half_time_scale, | | |
| 4789 | | user->divide_delay); | | |
| 4790 | | UNADJUST_DELAY(def_ptr->default_delay.typical, | | |
| 4791 | | user->time_scale, | | |
| 4792 | | user->half_time_scale, | | |
| 4793 | | user->divide_delay); | | |
| 4794 | | UNADJUST_DELAY(def_ptr->default_delay.maximum, | | |
| 4795 | | user->time_scale, | | |
| 4796 | | user->half_time_scale, | | |
| 4797 | | user->divide_delay); | | |
| 4798 | | } | | |
| 4799 | | } | | |
| 4800 | | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/util.c | DATE 5/23/89 | PAGE # 41/191 |
|--|--|--|-----------------|------------------|
| LINE # | | SOURCE TEXT | | |
| 4801 | | <pre> mtye_count = def_ptr->mtye_cnt; for (i = 0; i < mtye_count; ++i) { if ((long)def_ptr->mtye_table[i].minimum != -1) UNADJUST_DELAY(def_ptr->mtye_table[i].minimum, user->time_scale, user->half_time_scale, user->divide_delay); if ((long)def_ptr->mtye_table[i].typical != -1) UNADJUST_DELAY(def_ptr->mtye_table[i].typical, user->time_scale, user->half_time_scale, user->divide_delay); if ((long)def_ptr->mtye_table[i].maximum != -1) UNADJUST_DELAY(def_ptr->mtye_table[i].maximum, user->time_scale, user->half_time_scale, user->divide_delay); } } </pre> | | |
| 4820 | | <pre> } } </pre> | | |
| 4821 | | <pre> } } </pre> | | |
| 4822 | | <pre> } } </pre> | | |
| 4823 | | <pre> set_time_scale(user, number, units) USER_INFO *user; u_long number; long units; { double temp; switch (units) { case LM_FEMTOSECONDS: temp = 1000.0; break; case LM_PICOSECONDS: temp = 1.0; break; case LM_NANOSECONDS: temp = 0.001; break; case LM_MICROSECONDS: temp = 0.000001; break; default: lm_queue_message(ERROR_MSG, "internal simulator error: illegal unit: %d for simulation tick", units); return(FAILURE); } temp = temp / (double)number; if (temp > 1.0) { user->divide_delay = FALSE; user->time_scale = (u_long)temp; } else { user->divide_delay = TRUE; user->time_scale = (u_long)(1.0 / temp); } user->half_time_scale = user->time_scale / 2; /* Just in case */ if (user->time_scale == 0) user->time_scale = 1; return(SUCCESS); } </pre> | | |
| 4867 | | <pre> } } </pre> | | |
| 4868 | | <pre> } } </pre> | | |
| 4869 | | <pre> } } </pre> | | |
| 4870 | | <pre> } } </pre> | | |
| 4871 | | <pre> } } </pre> | | |
| 4872 | | <pre> } } </pre> | | |
| 4873 | | <pre> } } </pre> | | |
| 4874 | | <pre> } } </pre> | | |
| 4875 | | <pre> } } </pre> | | |
| 4876 | | <pre> } } </pre> | | |
| 4877 | | <pre> } } </pre> | | |
| 4878 | | <pre> } } </pre> | | |
| 4879 | | <pre> } } </pre> | | |
| 4880 | | <pre> } } </pre> | | |
| 4881 | | <pre> } } </pre> | | |
| 4882 | | <pre> } } </pre> | | |
| 4883 | | <pre> } } </pre> | | |
| 4884 | | <pre> } } </pre> | | |
| 4885 | | <pre> } } </pre> | | |
| 4886 | | <pre> } } </pre> | | |
| 4887 | | <pre> } } </pre> | | |
| 4888 | | <pre> } } </pre> | | |
| 4889 | | <pre> } } </pre> | | |
| 4890 | | <pre> } } </pre> | | |
| 4891 | | <pre> } } </pre> | | |
| 4892 | | <pre> } } </pre> | | |
| 4893 | | <pre> } } </pre> | | |
| 4894 | | <pre> } } </pre> | | |
| 4895 | | <pre> } } </pre> | | |
| 4896 | | <pre> } } </pre> | | |
| 4897 | | <pre> } } </pre> | | |
| 4898 | | <pre> } } </pre> | | |
| 4899 | | <pre> } } </pre> | | |
| 4900 | | <pre> } } </pre> | | |
| 4901 | | <pre> } } </pre> | | |
| 4902 | | <pre> } } </pre> | | |
| 4903 | | <pre> } } </pre> | | |
| 4904 | | <pre> } } </pre> | | |
| 4905 | | <pre> } } </pre> | | |
| 4906 | | <pre> } } </pre> | | |
| 4907 | | <pre> } } </pre> | | |
| 4908 | | <pre> } } </pre> | | |
| 4909 | | <pre> } } </pre> | | |
| 4910 | | <pre> } } </pre> | | |
| 4911 | | <pre> } } </pre> | | |
| 4912 | | <pre> } } </pre> | | |
| 4913 | | <pre> } } </pre> | | |
| 4914 | | <pre> } } </pre> | | |
| 4915 | | <pre> } } </pre> | | |
| 4916 | | <pre> } } </pre> | | |
| 4917 | | <pre> } } </pre> | | |
| 4918 | | <pre> } } </pre> | | |
| 4919 | | <pre> } } </pre> | | |
| 4920 | | <pre> } } </pre> | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM lm1000/util.c | DATE 5/23/89 TIME 6:14:53 pm | PAGE # 42/192 |
|--|--|--|---------------------------------------|------------------|
| LINE # | | SOURCE TEXT | | |
| 4921 | | while (i < CRYPTED_PASSWORD_LENGTH) { | | |
| 4922 | | temp_ptr = password; | | |
| 4923 | | while ((c = *temp_ptr++) && (i < CRYPTED_PASSWORD_LENGTH)) { | | |
| 4924 | | password_buffer[i++] = c; | | |
| 4925 | | } | | |
| 4926 | | } | | |
| 4927 | | for (i=0; i<CRYPTED_PASSWORD_LENGTH; i++) | | |
| 4928 | | crypt_string[i] = | | |
| 4929 | | (((char) i + strlen(password) << 3) ^ (password_buffer[i] ^ (password_buffer[CRYPTED_PASSWORD_LENGTH-(i+1)] << 1))); | | |
| 4930 | | } | | |
| 4931 | | return(SUCCESS); | | |
| 4932 | | } | | |
| 4933 | | } | | |
| 4934 | | } | | |
| 4935 | | check_password(user) | | |
| 4936 | | USER_INFO *user; | | |
| 4937 | | { | | |
| 4938 | | char good_password[CRYPTED_PASSWORD_LENGTH]; | | |
| 4939 | | u_long status; | | |
| 4940 | | u_char i; | | |
| 4941 | | DPRINTF(("inside check_password\n")); | | |
| 4942 | | if (user->good_password == TRUE) { | | |
| 4943 | | DPRINTF(("user->good_password is TRUE\n")); | | |
| 4944 | | return(SUCCESS); | | |
| 4945 | | } | | |
| 4946 | | if (proc_lm_rd(LM_NVRAM_MEMORY, 0, PASSWORD, | | |
| 4947 | | sizeof PASSWORD, good_password, &status) == FAILURE) { | | |
| 4948 | | lm_queue_message(ERROR_MSG, "internal error: Failed to read password"); | | |
| 4949 | | DPRINTF(("error in read\n")); | | |
| 4950 | | return(FAILURE); | | |
| 4951 | | } | | |
| 4952 | | /* Check for no password */ | | |
| 4953 | | for (i = 0; i < CRYPTED_PASSWORD_LENGTH; ++i) { | | |
| 4954 | | if (good_password[i] != '\0') | | |
| 4955 | | break; | | |
| 4956 | | } | | |
| 4957 | | if (i == CRYPTED_PASSWORD_LENGTH) { | | |
| 4958 | | DPRINTF(("No password is set\n")); | | |
| 4959 | | lm_queue_message(WARNING_MSG, "No password set on modeler"); | | |
| 4960 | | return(SUCCESS); | | |
| 4961 | | } | | |
| 4962 | | /* Check for CPU switch 3 */ | | |
| 4963 | | status = read_loc_long((u_long *)CPU_STATUS_REG_ADDR); | | |
| 4964 | | if (status & CPU_SW_SWITCH3_MASK) { | | |
| 4965 | | DPRINTF(("Password not checked due to hardware override\n")); | | |
| 4966 | | lm_queue_message(WARNING_MSG, "Password not checked due to hardware override"); | | |
| 4967 | | return(SUCCESS); | | |
| 4968 | | } | | |
| 4969 | | DPRINTF(("error in check_password\n")); | | |
| 4970 | | return(FAILURE); | | |
| 4971 | | } | | |
| 4972 | | } | | |
| 4973 | | write_password(encrypted_password) | | |
| 4974 | | char *encrypted_password; | | |
| 4975 | | { | | |
| 4976 | | u_long status; | | |
| 4977 | | DPRINTF(("inside write_password\n")); | | |
| 4978 | | if (proc_lm_wr(LM_NVRAM_MEMORY, 0, PASSWORD, | | |
| 4979 | | sizeof PASSWORD, encrypted_password, &status) == FAILURE) { | | |
| 4980 | | lm_queue_message(ERROR_MSG, "internal error: Failed to write password"); | | |
| 4981 | | return(FAILURE); | | |
| 4982 | | } | | |
| 4983 | | return(SUCCESS); | | |
| 4984 | | } | | |
| 4985 | | compare_password(encrypted_password) | | |
| 4986 | | char *encrypted_password; | | |
| 4987 | | { | | |
| 4988 | | char good_password[CRYPTED_PASSWORD_LENGTH]; | | |
| 4989 | | static char backdoor_password[] = { 0xD5, 0xCF, 0xD0, 0xED, 0xE0, 0xFA, | | |
| 4990 | | 0xE1, 0xCA, 0x15, 0x0F, 0x10, 0x1D, 0x20, 0x3A, 0x23, 0x0A }; | | |
| 4991 | | /* DO NOT change this unless you change lm_crypt() */ | | |
| 4992 | | u_long status; | | |
| 4993 | | u_char i; | | |
| 4994 | | DPRINTF(("inside compare_password\n")); | | |
| 4995 | | if (proc_lm_rd(LM_NVRAM_MEMORY, 0, PASSWORD, | | |
| 4996 | | sizeof PASSWORD, good_password, &status) == FAILURE) { | | |
| 4997 | | DPRINTF(("error in read\n")); | | |
| 4998 | | lm_queue_message(ERROR_MSG, "internal error: Failed to read password"); | | |
| 4999 | | return(FAILURE); | | |
| 5000 | | } | | |
| 5001 | | DPRINTF(("encrypted password: %s backdoor password: %s\n", | | |
| 5002 | | encrypted_password, backdoor_password)); | | |
| 5003 | | /* Check for backdoor passwords */ | | |
| 5004 | | if (memcmp(encrypted_password, | | |
| 5005 | | backdoor_password, CRYPTED_PASSWORD_LENGTH) == 0) { | | |
| 5006 | | DPRINTF(("password matches with backdoor password\n")); | | |
| 5007 | | return(SUCCESS); | | |
| 5008 | | } | | |
| 5009 | | /* Compare the password the user entered with the one in NVRAM */ | | |
| 5010 | | if (memcmp(encrypted_password, good_password, CRYPTED_PASSWORD_LENGTH) != 0) { | | |
| 5011 | | DPRINTF(("password doesn't match\n")); | | |
| 5012 | | lm_queue_message(ERROR_MSG, "Invalid password entered"); | | |
| 5013 | | return(FAILURE); | | |
| 5014 | | } | | |
| 5015 | | DPRINTF(("password matches !!!!!\n")); | | |
| 5016 | | return(SUCCESS); | | |
| 5017 | | } | | |
| 5018 | | } | | |
| 5019 | | } | | |
| 5020 | | } | | |
| 5021 | | } | | |
| 5022 | | } | | |
| 5023 | | } | | |
| 5024 | | } | | |
| 5025 | | } | | |
| 5026 | | } | | |
| 5027 | | } | | |
| 5028 | | } | | |
| 5029 | | } | | |
| 5030 | | } | | |
| 5031 | | } | | |
| 5032 | | } | | |
| 5033 | | } | | |
| 5034 | | } | | |

1707

5,353,243

1708

| | | | | |
|--|--|------------------------|---------|------------|
| Copyright 1989 Logic Modeling Systems | | FILE | DATE | PAGE # |
| | | bsp.lm1000/bsp.assem.s | 5/23/89 | 1/1 |
| | | | TIME | 4:41:10 pm |

| LINE # | TEXT |
|--------|---|
| 1 | SOCS ID: bsp.assem.s rev 3.1, 4/24/89 at 07:54:47 |
| 2 | ----- File Defines ----- |
| 3 | |
| 4 | ===== |
| 5 | DEFINITIONS SPECIFIC FOR THE CPU BOARD |
| 6 | ===== |
| 7 | DISINT = 0x3700 |
| 8 | ENAIPT = 0x1000 |
| 9 | RTSCOPE = 0x01 |
| 10 | VRTX = 0x00 |
| 11 | IVT_BASE = 0x000000 |
| 12 | DUART_RX = 0x10c20014 |
| 13 | STATUS_REG_A = 0x10c20004 |
| 14 | STATUS_REG_B = 0x10c20014 |
| 15 | DUART_RX_TX_REG_A = 0x10c2000c |
| 16 | DUART_RX_TX_REG_B = 0x10c2001c |
| 17 | CHAR_TX_A = 24 |
| 18 | CHAR_RX_A = 25 |
| 19 | CHAR_TX_B = 28 |
| 20 | CHAR_RX_B = 29 |
| 21 | CPU_INTX_REG = 0x10c60002 |
| 22 | CPU_MISC_CONTROL = 0x10c60001 |
| 23 | FALSE = 0 |
| 24 | TEST_LED = 0x40 |
| 25 | TIMER_CONTROL = 0x10c3000c |
| 26 | TIMER1_COUNT = 0x10c30008 |
| 27 | READ_COUNTER2 = 0x28000000 |
| 28 | TIMER2_TOTAL = 0x013E |
| 29 | TIMER2_LOW_COUNT = 0x40000000 |
| 30 | TIMER2_HIGH_COUNT = 0x01000000 |
| 31 | ENABLE_LANCE = 0x08 |
| 32 | CPU_FUSC_REG = 0x10c60001 |
| 33 | |
| 34 | ===== |
| 35 | VRTX and RTSCOPE function codes |
| 36 | ===== |
| 37 | TR_ENTER = 0x0102 |
| 38 | TR_EXIT = 0x0103 |
| 39 | TR_RICER = 0x0104 |
| 40 | TR_TIMER = 0x0105 |
| 41 | UI_TIMER = 0x0014 |
| 42 | UI_RICER = 0x0013 |
| 43 | UI_ENTER = 0x0016 |
| 44 | UI_EXIT = 0x0011 |
| 45 | UI_TIMER = 0x0012 |
| 46 | VRTX_INIT = 0x0030 |
| 47 | TR_INIT = 0x0100 |
| 48 | TR_GO = 0x0101 |
| 49 | SC_GO = 0x11 |
| 50 | SC_SPOST = 0x1e |
| 51 | SC_TCREATE = 0x00 |
| 52 | |
| 53 | ===== |
| 54 | MAIN TASK DEFINITIONS |
| 55 | ===== |
| 56 | MAIN_TASK = 0 User mode |
| 57 | MAIN_TID = 0 Task ID number 0 |
| 58 | MAIN_TPRI = 0 Task priority 0 |
| 59 | ----- End Defines ----- |
| 60 | |
| 61 | ===== |
| 62 | IOFF = 0x11 |
| 63 | ION = 0x11 |
| 64 | .data |
| 65 | .globl Wart_mask, _in_tick, _got_a_char, _debug_key, _debug_char |
| 66 | .globl _save_pc |
| 67 | .globl _bus_error_address Bus error reporting routine |
| 68 | _save_pc: .long 0 |
| 69 | _debug_key: .long 0 got a key for debugging. |
| 70 | _debug_char: .long 0 |
| 71 | _got_a_char: .long 0 Tell housekeeping of chars. |
| 72 | so it can print a message. |
| 73 | |
| 74 | _bus_error_address: .long 0 |
| 75 | _Wart_mask: .long 0 this is the DUART INTERRUPT MASK |
| 76 | _in_tick: .long 0 |
| 77 | _housekeeping: .word 0 Timer variable for the housekeeping task |
| 78 | _even |
| 79 | _IOFF: .byte 0 Keep track of ION/IOFF |
| 80 | |
| 81 | ===== |
| 82 | ENTRY POINT. When in EPROM, the following two longs are fetched |
| 83 | by the processor during a restart. |
| 84 | ===== |
| 85 | |
| 86 | .long 0x1FFFF0 Initial Stack Pointer |
| 87 | .long bsp_start Initial Program Counter |
| 88 | |
| 89 | ===== |
| 90 | |
| 91 | ----- Start of init code ----- |
| 92 | |
| 93 | .text |
| 94 | .globl bsp_start |
| 95 | bsp_start: |
| 96 | |
| 97 | movw \$DISINT, ar Set Interrupt Level |
| 98 | movl \$0x1FFFF0, sp Load Stack Pointer (for soft boot) |
| 99 | reset |
| 100 | movl \$IVT_BASE, d0 |
| 101 | d0, vdr Set Vector Base Register up |
| 102 | movl \$3, d0 |
| 103 | movc d0, cacr Clear and enable instruction cache |
| 104 | |
| 105 | jar _init_sys Exit to C code, never to return |
| 106 | |
| 107 | |
| 108 | |
| 109 | |
| 110 | |
| 111 | |
| 112 | |
| 113 | |
| 114 | |
| 115 | |
| 116 | |
| 117 | |
| 118 | |
| 119 | |
| 120 | |

| Copyright 1989 Logic Modeling Systems | | FILE bsp.lm1000/bsp.assem.s | DATE 5/23/89 TIME 4:41:10 pm | PAGE # 2/2 |
|--|---|--------------------------------|---------------------------------------|---------------|
| LINE # | TEXT | | | |
| 121 | Utility Routines called by the C Init code. | | | |
| 122 | | | | |
| 123 | | | | |
| 124 | | | | |
| 125 | | | | |
| 126 | | | | |
| 127 | ---- Vrtx Init Interface | | | |
| 128 | | | | |
| 129 | .globl _vrtx_init | | | |
| 130 | _vrtx_init: | | | |
| 131 | movl | \$VRTX_INIT, %d0 | Load Vrtx Init Function code | |
| 132 | trap | \$VRTX | Trap to Vrtx | |
| 133 | rti | | Return Error to C in D0 | |
| 134 | | | | |
| 135 | | | | |
| 136 | | | | |
| 137 | | | | |
| 138 | ---- Rtscope Init Interface | | | |
| 139 | | | | |
| 140 | .globl _rts_init | | | |
| 141 | _rts_init: | | | |
| 142 | link | %d0, %d0 | frame pointer | |
| 143 | movl | %d0, %sp | save A0 on the stack | |
| 144 | movl | \$_Rtscope_conf, %d0 | Rtscope conf. table addr | |
| 145 | movl | \$RTSCOPE_INIT, %d0 | DSINIT | |
| 146 | trap | \$RTSCOPE | call Rtscope | |
| 147 | movl | %sp, %d0 | restore A0 | |
| 148 | unlk | %d0 | restore A6 | |
| 149 | rti | | | |
| 150 | | | | |
| 151 | | | | |
| 152 | | | | |
| 153 | ---- Rtscope Go Interface | | | |
| 154 | | | | |
| 155 | .globl _rts_go | | | |
| 156 | _rts_go: | | | |
| 157 | cmpl | \$_Rtscope_present, %d0 | Is rtscope present | |
| 158 | jeq | \$_Rtscope_1, %d0 | | |
| 159 | movl | \$_SC_GO, %d0 | DMGO | |
| 160 | trap | \$RTSCOPE | call Rtscope | |
| 161 | no_rtscope_1: | | | |
| 162 | movl | \$_main, %d0 | New task address | |
| 163 | movl | \$_MAIN_THMODE, %d1 | Task mode | |
| 164 | movl | \$_MAIN_TID, %d2 | Task ID number | |
| 165 | movl | \$_MAIN_TPRI, %d3 | Task priority | |
| 166 | movl | \$_SC_TCREATE, %d0 | System call code | |
| 167 | trap | \$VRTX | Call VRTX | |
| 168 | | | | |
| 169 | ---- Issue VRTX GO system call | | | |
| 170 | | | | |
| 171 | movl | \$_SC_GO, %d0 | SC_GO | |
| 172 | trap | \$VRTX | Shouldn't come back | |
| 173 | rti | | | |
| 174 | | | | |
| 175 | bad_start: | | | |
| 176 | jmp | _bad_start | | |
| 177 | | | | |
| 178 | | | | |
| 179 | | | | |
| 180 | Interrupt Service Routines | | | |
| 181 | | | | |
| 182 | | | | |
| 183 | | | | |
| 184 | ---- Clock Board Error ISR | | | |
| 185 | | | | |
| 186 | .globl _error_isr, _parity_isr | | | |
| 187 | _error_isr: | | | |
| 188 | movl | %d0, %sp | | |
| 189 | movl | \$_K7ife, %sp | dont save a7 & d0 save d1-d7, a0-a6 | |
| 190 | movl | \$_K7ife, %sp | This is our C interrupt routine | |
| 191 | jar | \$_K7ife, %sp | restore above regs. | |
| 192 | movl | %sp, %d0 | | |
| 193 | | | | |
| 194 | ---- Perform UI_EXIT from ISR through Vrtx | | | |
| 195 | | | | |
| 196 | movl | \$_UI_EXIT, %d0 | | |
| 197 | trap | \$VRTX | | |
| 198 | | | | |
| 199 | ---- Parity ISR | | | |
| 200 | | | | |
| 201 | _parity_isr: | | | |
| 202 | movl | %d0, %sp | | |
| 203 | movl | \$_K7ife, %sp | dont save a7 & d0 save d1-d7, a0-a6 | |
| 204 | jar | \$_K7ife, %sp | This is our C interrupt routine | |
| 205 | movl | %sp, %d0 | restore above regs. | |
| 206 | | | | |
| 207 | ---- Perform UI_EXIT from ISR through Vrtx | | | |
| 208 | | | | |
| 209 | movl | \$_UI_EXIT, %d0 | | |
| 210 | trap | \$VRTX | | |
| 211 | | | | |
| 212 | ---- Ethernet ISR | | | |
| 213 | | | | |
| 214 | .globl _ether_isr | | | |
| 215 | _ether_isr: | | | |
| 216 | movl | %d0, %sp | | |
| 217 | movl | \$_K7ife, %sp | dont save a7 & d0 save d1-d7, a0-a6 | |
| 218 | movl | \$_system_call, %d0 | check if a system call is made? | |
| 219 | jar | \$_K7ife, %sp | This is our C interrupt routine | |
| 220 | movl | \$_system_call, %d0 | let us check bit 0 | |
| 221 | movl | %sp, %d0 | restore above regs. | |
| 222 | btst | %d0, %d0 | should we exit thru VRTX | |
| 223 | beq | \$_no_sys_calls, %d0 | go to no system calls, if no VRTX | |
| 224 | | | | |
| 225 | | | | |
| 226 | ---- Perform UI_EXIT from ISR through Vrtx | | | |
| 227 | | | | |
| 228 | movl | \$_UI_EXIT, %d0 | | |
| 229 | trap | \$VRTX | | |
| 230 | | | | |
| 231 | ---- NO SYSTEM CALLS WERE MADE | | | |
| 232 | | | | |
| 233 | _no_sys_calls: | | | |
| 234 | movl | %sp, %d0 | | |
| 235 | rti | | | |
| 236 | | | | |
| 237 | ---- DUART ISR | | | |
| 238 | Channel A Serial Port - RX buffer full | | | |
| 239 | | | | |
| 240 | .globl _serial_isr | | | |

| | | | | |
|--|---|--------------------------------|-----------------|---|
| Copyright 1982 Logic Modeling Systems | | FILE bsp.lm1000/bsp.assem.s | DATE 5/23/89 | PAGE # 3/3 |
| TIME 4:41:10 pm | | | | |
| LINE # | TEXT | | | |
| 241 | _serial_isr: | | | |
| 242 | movl \$0, %eax | | | |
| 243 | movl \$1, %eax | | | |
| 244 | movl \$0, %eax | | | |
| 245 | movl \$UART_SR, %eax | | | |
| 246 | movl %eax, %di | | | get status |
| 247 | andl %uart_mask, %di | | | load into reg |
| 248 | andl \$0x000000, %di | | | only look @ interrupts we are interested in |
| 249 | jqc %rtscope_serial_isr | | | see if VRTX interrupt |
| 250 | movl %eax, %di | | | not go to RTscope |
| 251 | btst %CHAR_RX_A, %di | | | load into reg |
| 252 | bqc %not_rx_a | | | check for CHAR_RX_A |
| 253 | movl \$STATUS_REG_A, %eax | | | nothing rxd on channel a |
| 254 | movl %eax, %eax | | | get the status |
| 255 | movl \$UART_RX_TX_REG_A, %eax | | | status is in 24-31 |
| 256 | movl %eax, %di | | | get the byte |
| 257 | andl \$0x00000000, %di | | | byte is in 24-31 |
| 258 | jeq %not_rx_a | | | see if error to rxd, break or framing |
| 259 | movl \$24, %eax | | | false alarm, there was an error |
| 260 | sarl %eax, %di | | | shift 24 bits |
| 261 | cmpl \$DOFF, %di | | | dl bits 0-7 = data |
| 262 | jeq %not_xoff | | | Flow control check for RXN & XOFF |
| 263 | movb %DOFF, %eax | | | |
| 264 | jar %tx_exit | | | |
| 265 | hrr %not_tx_a | | | |
| 266 | not_xoff: | | | |
| 267 | cmpl %ECN, %di | | | |
| 268 | jeq %not_xon | | | |
| 269 | movb %ECN, %eax | | | |
| 270 | jar %tx_exit | | | |
| 271 | hrr %not_tx_a | | | |
| 272 | not_xon: | | | |
| 273 | movl \$1, %eax | | | |
| 274 | movl \$1, %eax | | | we got a char tell em' we are busy |
| 275 | movl %di, %eax | | | got a key for debugging. |
| 276 | andl %not_debug_char, %eax | | | |
| 277 | movl %eax, %eax | | | |
| 278 | trap \$VRTX | | | |
| 279 | not_tx_a: | | | |
| 280 | movl \$UART_SR, %eax | | | get status |
| 281 | movl %eax, %di | | | load into reg |
| 282 | btst %CHAR_TX_A, %di | | | check for CHAR_TX_A |
| 283 | bqc %not_tx_a | | | nothing to tx on channel a |
| 284 | jar %tx_exit | | | go and transmit |
| 285 | not_tx_a: | | | |
| 286 | movl %eax, %eax | | | restore address register |
| 287 | movl %eax, %di | | | |
| 288 | movl \$UI_EXIT, %eax | | | |
| 289 | trap \$VRTX | | | |
| 290 | END %tx_isr | | | |
| 291 | Channel A Serial Port - TX buffer empty | | | |
| 292 | global %tx_isr, %vtx_tx | | | |
| 293 | %tx_isr: | | | |
| 294 | cmpl \$UI_EXIT, %eax | | | Check flow control is Not on |
| 295 | jeq %no_flow_ctrl | | | |
| 296 | hrr %tx_exit | | | |
| 297 | no_flow_ctrl: | | | |
| 298 | movl \$UI_TXRDY, %eax | | | |
| 299 | trap \$VRTX | | | |
| 300 | cmpl \$0, %eax | | | |
| 301 | beq %tx_exit | | | |
| 302 | ----- This is the VRTX TX routine ----- | | | |
| 303 | ----- di = data to TX ----- | | | |
| 304 | ----- save A0 & B0 ----- | | | |
| 305 | ----- This is the hook for VRTX to resume or start transmitting chars ----- | | | |
| 306 | _vtx_tx: | | | |
| 307 | movl \$0, %eax | | | |
| 308 | movl \$UART_RX_TX_REG_A, %eax | | | get address of register |
| 309 | movl \$24, %eax | | | shift 24 bits to get byte in upper bits |
| 310 | sarl %eax, %di | | | dl bits 00-07 = data |
| 311 | movl %di, %eax | | | byte is in 24-31 after shift |
| 312 | movl \$UART_SR, %eax | | | get status reg contents |
| 313 | movl %uart_mask, %di | | | this is a write only reg. so read from memory |
| 314 | andl \$0x00000000, %di | | | dl = 03 enable tx ints |
| 315 | movl %di, %uart_mask | | | save in memory |
| 316 | movl %eax, %eax | | | load into reg |
| 317 | movl %eax, %di | | | restore registers |
| 318 | rti | | | |
| 319 | tx_exit: | | | |
| 320 | movl \$UART_SR, %eax | | | get status reg contents |
| 321 | movl %uart_mask, %di | | | this is a write only reg. so read from memory |
| 322 | andl \$0x00000000, %di | | | dl = disable tx ints |
| 323 | movl %di, %uart_mask | | | save in memory |
| 324 | movl %eax, %eax | | | load into reg |
| 325 | rti | | | |
| 326 | END %tx_isr | | | |
| 327 | RTscope Serial Int. on chan B | | | |
| 328 | rtscope_serial_isr: | | | |
| 329 | movl %eax, %di | | | load into reg |
| 330 | btst %CHAR_RX_B, %di | | | check for CHAR_RX_B |
| 331 | bqc %not_rx_b | | | nothing rxd on channel b |
| 332 | movl \$UART_RX_TX_REG_B, %eax | | | get the byte |
| 333 | movl %eax, %di | | | byte is in 24-31 |
| 334 | cmpl \$0, %RTscope_present | | | is RTscope present |
| 335 | jeq %not_rx_b | | | |
| 336 | movl \$24, %eax | | | |
| 337 | sarl %eax, %di | | | shift 24 bits |
| 338 | cmpl \$29, %di | | | dl bits 0-7 = data |
| 339 | jeq %not_cache_disable | | | check for " |
| 340 | movl \$8, %eax | | | cache disable |
| 341 | movl %eax, %eax | | | |
| 342 | jmp %not_rx_b | | | Clear and disable instruction cache |
| 343 | not_cache_disable: | | | |
| 344 | movl \$RTSCOPE, %eax | | | |
| 345 | trap \$VRTX | | | |
| 346 | not_rx_b: | | | |
| 347 | movl \$UART_SR, %eax | | | get status |

| Copyright 1989 Logic Modeling Systems | | FILE bsp.lm1000/bsp.assem.s | DATE 5/23/89 TIME 4:41:10 pm | PAGE # 4/4 |
|--|--|--|---|---------------|
| LINE # | TEXT | | | |
| 361 | movl | a0,d1 | load into reg | |
| 362 | btest | \$CHAR_TX_B,d1 | check for CHAR_TX_B | |
| 363 | beq | not_tx_b | nothing can be transmitted on channel b | |
| 364 | jar | txb_ixr | no and transmit | |
| 365 | not_tx_b: | | | |
| 366 | movl | sp0,a0 | restore address register | |
| 367 | movl | sp0,d1 | | |
| 368 | | | | |
| 369 | cmpl | \$0,_rtscope_present | Is rtscope present | |
| 370 | jeq | no_rtscope_1 | | |
| 371 | movl | \$RTX_EXIT,d0 | | |
| 372 | trap | RTXSCOPE | | |
| 373 | no_rtscope_1: | | | |
| 374 | movl | \$RTX_EXIT,d0 | | |
| 375 | trap | RTX | | |
| 376 | | | | |
| 377 | ----- | END txb_ixr | | |
| 378 | | | | |
| 379 | ----- | Channel B Serial Port - TX buffer empty | | |
| 380 | | | | |
| 381 | | global txb_ixr, _vrtx_txb | | |
| 382 | txb_ixr: | | | |
| 383 | | | | |
| 384 | cmpl | \$0,_rtscope_present | Is rtscope present | |
| 385 | jeq | txb_exit | | |
| 386 | movl | \$RTX_EXIT,d0 | | |
| 387 | trap | RTXSCOPE | | |
| 388 | cmpl | \$0,d0 | | |
| 389 | beq | txb_exit | | |
| 390 | | | | |
| 391 | | | | |
| 392 | ----- | This is the RTXSCOPE TX routine | | |
| 393 | ----- | d1 = data to TX | | |
| 394 | ----- | save A0 & D0 | | |
| 395 | ----- | This is the hook for RTXSCOPE to resume or start transmitting chars -- | | |
| 396 | | | | |
| 397 | vrtx_txb: | | | |
| 398 | movl | d0,sp0 | get address of register | |
| 399 | movl | sp0,sp0 | shift 24-bits to get byte in upper bits | |
| 400 | movl | \$UART_RX_TX_REG_B,a0 | d1 bits 00-07 = data | |
| 401 | moveq | \$24,d0 | byte is in 24-31 after shift | |
| 402 | asll | d0,d1 | get status reg contents | |
| 403 | movl | d1,sp0 | this is a write only reg. so read from memory | |
| 404 | movl | \$UART_SR,a0 | d1 = 01 enable tx lists | |
| 405 | movl | \$uart_mask,d1 | save in memory | |
| 406 | orl | \$0x10000000,d1 | load into reg | |
| 407 | movl | d1,_uart_mask | restore registers | |
| 408 | movl | d1,sp0 | | |
| 409 | movl | sp0,a0 | | |
| 410 | movl | sp0,d0 | | |
| 411 | rts | | | |
| 412 | txb_exit: | | | |
| 413 | movl | \$UART_SR,a0 | get status reg contents | |
| 414 | movl | \$uart_mask,d1 | this is a write only reg. so read from memory | |
| 415 | andl | \$0x10000000,d1 | d1 = disable tx lists | |
| 416 | movl | d1,_uart_mask | save in memory | |
| 417 | movl | d1,sp0 | load into reg | |
| 418 | rts | | | |
| 419 | ----- | END txb_ixr | | |
| 420 | | | | |
| 421 | global _rtscope_is_poll, _rtscope_out_poll | | | |
| 422 | _rtscope_is_poll: | | | |
| 423 | movl | a0,sp0 | | |
| 424 | movl | d1,sp0 | | |
| 425 | movl | \$0x10000000,d0 | status | |
| 426 | cmpl | \$500,_auto_start | | |
| 427 | bit | inc_not_rx_b_poll | | |
| 428 | cmpl | \$500,_auto_start | | |
| 429 | beq | not_time_for_C | | |
| 430 | movl | \$0x57,d0 | | |
| 431 | bne | inc_not_rx_b_poll | | |
| 432 | not_time_for_C: | | | |
| 433 | cmpl | \$501,_auto_start | | |
| 434 | beq | not_time_for_O | | |
| 435 | movl | \$0x52,d0 | | |
| 436 | bne | inc_not_rx_b_poll | | |
| 437 | not_time_for_O: | | | |
| 438 | cmpl | \$502,_auto_start | | |
| 439 | beq | not_time_for_0xb | | |
| 440 | movl | \$0x00,d0 | | |
| 441 | bne | inc_not_rx_b_poll | | |
| 442 | not_time_for_0xb: | | | |
| 443 | movl | \$UART_SR,a0 | get status reg contents | |
| 444 | movl | a0,d1 | load into reg | |
| 445 | btest | \$CHAR_RX_B,d1 | check for CHAR_RX_B | |
| 446 | beq | not_rx_b_poll | nothing rcvd on channel b | |
| 447 | movl | \$UART_RX_TX_REG_B,a0 | get the byte | |
| 448 | movl | a0,d0 | byte is in 24-31 | |
| 449 | moveq | \$24,d1 | shift 24 bits | |
| 450 | asrl | d1,d0 | d1 bits 0-7 = data | |
| 451 | | | | |
| 452 | cmpl | \$29,d0 | check for " | |
| 453 | jbe | not_poll_osche_disable | cache disable | |
| 454 | movl | \$8,d0 | | |
| 455 | movc | d0,cacr | Clear and disable instruction cache | |
| 456 | jmp | not_rx_b | | |
| 457 | not_poll_osche_disable: | | | |
| 458 | | | | |
| 459 | not_rx_b_poll: | | | |
| 460 | movl | sp0,d1 | restore registers | |
| 461 | movl | sp0,a0 | | |
| 462 | rts | | | |
| 463 | inc_not_rx_b_poll: | | | |
| 464 | addl | \$1,_auto_start | | |
| 465 | movl | sp0,d1 | restore registers | |
| 466 | movl | sp0,a0 | | |
| 467 | rts | | | |
| 468 | | | | |
| 469 | | | | |
| 470 | | | | |
| 471 | _rtscope_out_poll: | | | |
| 472 | movb | sp0(7),d0 | | |
| 473 | movl | d1,sp0 | | |
| 474 | movl | a0,sp0 | | |
| 475 | movl | \$UART_SR,a0 | get status | |
| 476 | movl | a0,d1 | load into reg | |
| 477 | btest | \$CHAR_TX_B,d1 | check for CHAR_TX_B | |
| 478 | beq | not_tx_b_poll | nothing can be transmitted on channel b | |
| 479 | movl | \$UART_RX_TX_REG_B,a0 | get address of register | |
| 480 | moveq | \$24,d1 | shift 24 bits to get byte in upper bits | |

| | | | | |
|--|--|--------------------------------|---|---------------|
| Copyright 1989 Logic Modeling Systems | | FILE bsp.lm1000/bsp.assem.s | DATE 5/23/89 | PAGE # 5/5 |
| TEXT | | | | |
| LINE # | | | | |
| 481 | call | d1,d0 | d0 bits 00-07 = data | |
| 482 | movl | d0,a0@ | byte is in 24-31 after shift | |
| 483 | movl | \$UART_SR,a0 | get status reg contents | |
| 484 | movl | UART_mask,d1 | this is a write only reg. so read from memory | |
| 485 | orl | \$0x1000000,d1 | d1 = 01 enable tx intx | |
| 486 | movl | d1,UART_mask | save in memory | |
| 487 | movl | d1,a0@ | load into reg | |
| 488 | movl | \$0,d0 | successfully tx | |
| 489 | movl | spc+,a0 | restore registers | |
| 490 | movl | spc+,d1 | | |
| 491 | ret | | | |
| 492 | not_tx_poll: | | | |
| 493 | movl | \$0xffffffff,d0 | no tx took place | |
| 494 | movl | spc+,a0 | restore registers | |
| 495 | movl | spc+,d1 | | |
| 496 | ret | | | |
| 497 | | | | |
| 498 | ----- This is the BUS ERROR TX routine ----- | | | |
| 499 | ----- d1 = data to TX ----- | | | |
| 500 | ----- save A0 & D0 ----- | | | |
| 501 | | | | |
| 502 | globl | bus_error_tx | | |
| 503 | bus_error_tx: | | | |
| 504 | link | a6,\$0 | | |
| 505 | movl | d1,spc- | | |
| 506 | movl | d0,spc- | | |
| 507 | movl | a0,spc- | | |
| 508 | not_ready_tx_a: | | | |
| 509 | movl | \$UART_SR,a0 | get status | |
| 510 | movl | a0,d1 | load into reg | |
| 511 | btest | \$CHAR_TX_A,d1 | check for CHAR_TX_A | |
| 512 | beq | not_ready_tx_a | nothing to tx on Channel a | |
| 513 | movl | a0(\$1),d1 | data to tx | |
| 514 | movl | \$UART_TX_REG_A,a0 | get address of register | |
| 515 | movl | \$24,d0 | shift 24 bits to get byte in upper bits | |
| 516 | call | d0,d1 | d1 bits 00-07 = data | |
| 517 | movl | d1,a0@ | byte is in 24-31 after shift | |
| 518 | movl | spc+,a0 | restore registers | |
| 519 | movl | spc+,d0 | | |
| 520 | movl | spc+,d1 | | |
| 521 | unlk | a6 | | |
| 522 | ret | | | |
| 523 | ----- Countax/Timer Interrupt ----- | | | |
| 524 | | | | |
| 525 | globl | ct_isr | | |
| 526 | ct_isr: | | | |
| 527 | | | | |
| 528 | | | | |
| 529 | | | | |
| 530 | | | | |
| 531 | | | | |
| 532 | | | | |
| 533 | | | | |
| 534 | | | | |
| 535 | | | | |
| 536 | | | | |
| 537 | | | | |
| 538 | | | | |
| 539 | | | | |
| 540 | | | | |
| 541 | | | | |
| 542 | | | | |
| 543 | | | | |
| 544 | | | | |
| 545 | | | | |
| 546 | | | | |
| 547 | | | | |
| 548 | | | | |
| 549 | | | | |
| 550 | | | | |
| 551 | | | | |
| 552 | | | | |
| 553 | | | | |
| 554 | | | | |
| 555 | | | | |
| 556 | | | | |
| 557 | | | | |
| 558 | | | | |
| 559 | | | | |
| 560 | | | | |
| 561 | | | | |
| 562 | | | | |
| 563 | | | | |
| 564 | | | | |
| 565 | | | | |
| 566 | | | | |
| 567 | | | | |
| 568 | | | | |
| 569 | | | | |
| 570 | | | | |
| 571 | | | | |
| 572 | | | | |
| 573 | | | | |
| 574 | | | | |
| 575 | | | | |
| 576 | | | | |
| 577 | | | | |
| 578 | | | | |
| 579 | | | | |
| 580 | | | | |
| 581 | | | | |
| 582 | | | | |
| 583 | | | | |
| 584 | | | | |
| 585 | | | | |
| 586 | | | | |
| 587 | | | | |
| 588 | | | | |
| 589 | | | | |
| 590 | | | | |
| 591 | | | | |
| 592 | | | | |
| 593 | | | | |
| 594 | | | | |
| 595 | | | | |
| 596 | | | | |
| 597 | | | | |
| 598 | | | | |
| 599 | | | | |
| 600 | | | | |

| | | | | | |
|--|--|--------------------------------|---|--------------------|---------------|
| Copyright 1989 Logic Modeling Systems | | FILE bsp.lm1000/bsp.assem.s | # | DATE 5/23/89 | PAGE # 6/6 |
| | | | | TIME 4:41:10 pm | |

| LINE # | TEXT |
|--------|--|
| 601 | |
| 602 | ----- Perform UI_EXIT from ISR through Vrtx |
| 603 | |
| 604 | movl \$UI_EXIT,d0 |
| 605 | trap \$VRTX |
| 606 | |
| 607 | ----- END of_isr ----- |
| 608 | |
| 609 | ----- BUS ERROR HANDLER ----- |
| 610 | |
| 611 | |
| 612 | .globl _bus_isr |
| 613 | _bus_isr: |
| 614 | movl sp, _bus_error_address |
| 615 | |
| 616 | moveml \$0xffff,sp+ dont save a7 save d1-d7,a0-a6 |
| 617 | jar _bus_error |
| 618 | moveml sp+,\$0xffff restore above regs. |
| 619 | rts |
| 620 | |
| 621 | ===== |
| 622 | ----- TRAP ROUTINES for RTscope and VRTX |
| 623 | |
| 624 | ===== |
| 625 | |
| 626 | .globl _trap_vrtx |
| 627 | _trap_vrtx: |
| 628 | |
| 629 | trap \$VRTX |
| 630 | rts |
| 631 | |
| 632 | .globl _trap_rtscope |
| 633 | _trap_rtscope: |
| 634 | |
| 635 | trap \$RTSCOPE |
| 636 | rts |
| 637 | |
| 638 | |
| 639 | ===== |
| 640 | |
| 641 | ----- RESET_BRD |
| 642 | cause a board reset. |
| 643 | |
| 644 | |
| 645 | ===== |
| 646 | |
| 647 | .globl _reset_brd |
| 648 | _reset_brd: |
| 649 | reset |
| 650 | jar bsp_start |
| 651 | |
| 652 | board reset occurs here |
| 653 | |
| 654 | rts |
| 655 | |
| 656 | |
| 657 | .globl _do_nothing |
| 658 | _do_nothing: |
| 659 | rts |
| 660 | |
| 661 | |
| 662 | ----- Create a supervisory task ----- |
| 663 | err = sc_tcreate_supr_net_boot(task_address, SUPV_USER task , task_id, task priority) |
| 664 | if(err != 0) FAILURE; |
| 665 | |
| 666 | .globl _sc_tcreate_supr_net_boot |
| 667 | _sc_tcreate_supr_net_boot: |
| 668 | link a6,#0 |
| 669 | movl a0,sp+0 |
| 670 | movl d1,sp+4 |
| 671 | movl d2,sp+8 |
| 672 | movl d3,sp+12 |
| 673 | movl \$0,d1 |
| 674 | movl \$0,d1 |
| 675 | movl a6(8),a0 task address |
| 676 | movl a6(0xc),d1 task mode |
| 677 | movl a6(0x10),d1 task id |
| 678 | movl a6(0x14),d1 task priority |
| 679 | movl \$SC_TCREATE,d0 system call code |
| 680 | trap \$VRTX Call VRTX |
| 681 | movl sp+,d1 |
| 682 | movl sp+,d2 |
| 683 | movl sp+,d1 |
| 684 | movl sp+,a0 |
| 685 | unlk a6 |
| 686 | rts |
| 687 | |
| 688 | .globl _disable_cache |
| 689 | _disable_cache: |
| 690 | movl \$8,d0 |
| 691 | movc d0,cacr Clear and disable instruction cache |
| 692 | rts |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM bsp.lm1000/bsp.c | DATE 5/23/89 | PAGE # 1/7 |
|--|--|---|-----------------|---------------|
| LINE # | | SOURCE TEXT | | |
| 1 | | /* SOCS_ID: bsp.c rev 3.1.1, 4/24/89 at 07:54:50 */ | | |
| 2 | | /* | | |
| 3 | | /* This is part of the board support package | | |
| 4 | | /* Make sure that both bsp.lm1000/bsp.c and bsp.dia2a/bsp.c | | |
| 5 | | /* are exactly the same. | | |
| 6 | | /* | | |
| 7 | | /*include "common.h" | | |
| 8 | | /*include "task.h" | | |
| 9 | | /*include "uart.h" | | |
| 10 | | /*include "lm1000_wr.h" | | |
| 11 | | /*include "lm1000_r.h" | | |
| 12 | | /*include "cpu.h" | | |
| 13 | | /*include "cpu_voc.h" | | |
| 14 | | /*include "mod_err.h" | | |
| 15 | | /*include "nvram.h" | | |
| 16 | | /*include "id.h" | | |
| 17 | | /*----- External routine definitions -----*/ | | |
| 18 | | extern bus_iar(); | | |
| 19 | | extern serial_iar(); | | |
| 20 | | extern parity_iar(); | | |
| 21 | | extern ct_iar(); | | |
| 22 | | extern void reset_cpu(); | | |
| 23 | | extern ether_iar(); | | |
| 24 | | extern error_iar(); | | |
| 25 | | extern vrtx_tmr(); | | |
| 26 | | extern vrtx_tmr(); | | |
| 27 | | extern rtoscope_in_poll(); | | |
| 28 | | extern rtoscope_out_poll(); | | |
| 29 | | extern void id_load(); | | |
| 30 | | extern trp_vrtx(); | | |
| 31 | | extern trp_rtscope(); | | |
| 32 | | extern u_short lm_nvram_access(); | | |
| 33 | | extern u_long network_timeout; | | |
| 34 | | extern reset_hnd(); | | |
| 35 | | extern end; | | |
| 36 | | #define MSEC2NS 1000 | | |
| 37 | | /*----- Global variables -----*/ | | |
| 38 | | /* | | |
| 39 | | /* Initialize VRTX and RTSCOPE | | |
| 40 | | /* Activate RTSCOPE and VRTX. | | |
| 41 | | /* | | |
| 42 | | /* 1. Setup interrupt vector table | | |
| 43 | | /* 2. Create VRTX conf. table | | |
| 44 | | /* 3. Create RTSCOPE conf. table | | |
| 45 | | /* 4. Create component table | | |
| 46 | | /* 5. Initialize VRTX | | |
| 47 | | /* 6. Initialize RTSCOPE | | |
| 48 | | /* 7. Setup 8254 timers | | |
| 49 | | /* 8. Initialize SMART chan A | | |
| 50 | | /* 9. Enable CPU Regs | | |
| 51 | | /* 10. Partitions, create & malloc | | |
| 52 | | /* 11. RTSCOPE go command | | |
| 53 | | /*-----*/ | | |
| 54 | | /* unsigned long rtoscope = 0; | | |
| 55 | | /* u_long rtoscope_preset = 0; | | |
| 56 | | /* u_long total_malloc_size = 0; | | |
| 57 | | /* u_long available_malloc_size; | | |
| 58 | | /* ID FROM CPU id_prom; | | |
| 59 | | /* | | |
| 60 | | /* initialize routine | | |
| 61 | | /* | | |
| 62 | | init_sys() | | |
| 63 | | { | | |
| 64 | | long err = 0; | | |
| 65 | | char baud; | | |
| 66 | | u_short short_network_timeout; | | |
| 67 | | cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG; | | |
| 68 | | { | | |
| 69 | | RTSCOPE_preset = ((u_long *) (RTS_BASE)); | | |
| 70 | | /* | | |
| 71 | | /* set modeler state to booted | | |
| 72 | | /* | | |
| 73 | | modeler_state = BOOTED; | | |
| 74 | | /* | | |
| 75 | | /* Turn off load led | | |
| 76 | | /* | | |
| 77 | | p_cpu_ctl_reg->not_load_led = not_LED_OFF; | | |
| 78 | | /* | | |
| 79 | | /* shutdown and reset lanes. | | |
| 80 | | p_cpu_ctl_reg->not_lane_reset = 1; | | |
| 81 | | /* | | |
| 82 | | shutdown_and_reset_lanes(); | | |
| 83 | | /* | | |
| 84 | | /* calculate total partition size | | |
| 85 | | /* | | |
| 86 | | total_malloc_size = (u_long)CPU_RAM_SIZE - (u_long)(load); | | |
| 87 | | available_malloc_size = total_malloc_size; | | |
| 88 | | { | | |
| 89 | | setup_vrtx(); /* Setup the interrupt table */ | | |
| 90 | | init_vrtxble(); /* Create VRTX conf. table */ | | |
| 91 | | init_rtsble(); /* Create RTSCOPE conf. table */ | | |
| 92 | | init_cvt(); /* Create component table */ | | |
| 93 | | err = vrtx_init(); /* Init VRTX */ | | |
| 94 | | if (err) /* error, wait in loop */ | | |
| 95 | | { | | |
| 96 | | for(;;) | | |
| 97 | | { | | |
| 98 | | reset_cpu(SUICIDE); | | |
| 99 | | } | | |
| 100 | | /* Init RTSCOPE */ | | |
| 101 | | { | | |
| 102 | | if (RTSCOPE_preset) | | |
| 103 | | { | | |
| 104 | | err = rts_init(&(cvtble->r_cvt)); | | |
| 105 | | if (err) /* error, wait in loop */ | | |
| 106 | | { | | |
| 107 | | for(;;) | | |
| 108 | | { | | |
| 109 | | reset_cpu(SUICIDE); | | |
| 110 | | } | | |
| 111 | | } | | |
| 112 | | } | | |
| 113 | | /* | | |
| 114 | | /* Initialize board devices | | |
| 115 | | /* | | |
| 116 | | init_all_timer(); /* initialize all 3 timers */ | | |
| 117 | | /* | | |
| 118 | | /* validate NVRAM, setup baud rate, it should be valid | | |
| 119 | | /* | | |
| 120 | | /* | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM bsp.lm1000/bsp.c | DATE 5/23/89 | PAGE # 2/8 |
|--|--|--|-----------------|---------------|
| LINE # | | SOURCE TEXT | | |
| 121 | | (void)lm_svarm_access((char *)0, (u_long)0, (u_long)0, MEMORY_VALIDATE, (u_long *) &err); | | |
| 122 | | /* Modeler state to be read */ | | |
| 123 | | /* | | |
| 124 | | (void)lm_svarm_access(modeler_state, MODELER_STATE, sizeof(MODELER_STATE), MEMORY_WRITE, (u_long *) &err); | | |
| 125 | | Varta_isr(); | | |
| 126 | | Varta_isr(); | | |
| 127 | | /* | | |
| 128 | | /* net network timeout */ | | |
| 129 | | /* | | |
| 130 | | if(lm_svarm_access(&short_network_timeout, NETWORK_TIMEOUT, sizeof(NETWORK_TIMEOUT), MEMORY_READ, (u_long *) &err) == FAILURE) | | |
| 131 | | { | | |
| 132 | | sys_out("Unable to read network timeout\n"); | | |
| 133 | | } | | |
| 134 | | /* | | |
| 135 | | /* Turn seconds to milliseconds */ | | |
| 136 | | /* | | |
| 137 | | network_timeout = short_network_timeout; | | |
| 138 | | network_timeout *= MSECSECS_PER_SECOND; | | |
| 139 | | /* | | |
| 140 | | /* read id from */ | | |
| 141 | | /* | | |
| 142 | | id_load((u_char *)CPU_ID_PROM, (u_char *)(&id_prom)); /* fetch id prom */ | | |
| 143 | | enable_cpu_regs(); /* enable CPU regs */ | | |
| 144 | | rts_go(); /* Execute Kscope GO command */ | | |
| 145 | | /* | | |
| 146 | | /* | | |
| 147 | | /* SHOULD NOT COME BACK, if it does ?????????? */ | | |
| 148 | | /* do nothing */ | | |
| 149 | | /* | | |
| 150 | | /* | | |
| 151 | | { | | |
| 152 | | reset_cpu(SWICIDE); | | |
| 153 | | } | | |
| 154 | | /* end init_sys */ | | |
| 155 | | /* | | |
| 156 | | /* | | |
| 157 | | /* | | |
| 158 | | /* Create VRTX configuration table */ | | |
| 159 | | /* | | |
| 160 | | /* | | |
| 161 | | init_vrtxtbl() | | |
| 162 | | { | | |
| 163 | | /* | | |
| 164 | | VRTX_CNTRL *ct = (VRTX_CNTRL *)(&(cftbl->v_cft)); | | |
| 165 | | /* | | |
| 166 | | /* | | |
| 167 | | ct->v_wsapc = VRTX_WK_SPACE; /* VRTX workspace address */ | | |
| 168 | | ct->v_ws_size = VRTX_WK_SIZE; /* VRTX workspace size */ | | |
| 169 | | ct->sys_stk_size = SYS_STK; /* System stack size */ | | |
| 170 | | ct->isr_stk_size = ISR_STK; /* Interrupt stack size */ | | |
| 171 | | ct->ch_count = CH_COUNT; /* Number of VRTX ch */ | | |
| 172 | | ct->real = 0; /* Reserved */ | | |
| 173 | | ct->real = 0; /* Reserved */ | | |
| 174 | | ct->c_dis_lev = VC_DIS_LEVEL; /* Component disable level */ | | |
| 175 | | ct->usr_stk_size = USR_STK; /* User stack size */ | | |
| 176 | | ct->real = 0; /* Reserved */ | | |
| 177 | | ct->usr_tsk_count = TASK_COUNT; /* Number of tasks */ | | |
| 178 | | ct->real = 0; /* Reserved */ | | |
| 179 | | ct->tx_rdy = (long) Vrtx_tx; /* Transmit ready interrupt */ | | |
| 180 | | ct->tx_rdy = 0; /* User supplied txready */ | | |
| 181 | | ct->tx_rdy = 0; /* User supplied txready */ | | |
| 182 | | ct->tx_rdy = 0; /* User supplied txready */ | | |
| 183 | | if(Kscope_present) | | |
| 184 | | ct->cvt_add = (CVT_TBL *)(&(cftbl->cvt)); /* Component conf. table add */ | | |
| 185 | | else | | |
| 186 | | ct->cvt_add = (CVT_TBL *) 0; /* Component conf. table add */ | | |
| 187 | | /* | | |
| 188 | | /* end init_vrtxtbl */ | | |
| 189 | | /* | | |
| 190 | | /* | | |
| 191 | | /* | | |
| 192 | | /* | | |
| 193 | | /* Create Kscope configuration table */ | | |
| 194 | | /* | | |
| 195 | | /* | | |
| 196 | | init_ksctbl() | | |
| 197 | | { | | |
| 198 | | /* | | |
| 199 | | /* | | |
| 200 | | KSCPT_CNTRL *ct = (KSCPT_CNTRL *)(&(cftbl->ks_cft)); | | |
| 201 | | KSCPT_CNTRL *ct = (KSCPT_CNTRL *)(&(cftbl->ks_cft)); | | |
| 202 | | short | | |
| 203 | | /* | | |
| 204 | | /* Kscope configuration table */ | | |
| 205 | | /* | | |
| 206 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 207 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 208 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 209 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 210 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 211 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 212 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 213 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 214 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 215 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 216 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 217 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 218 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 219 | | ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */ | | |
| 220 | | /* | | |
| 221 | | /* Kscope IO configuration table */ | | |
| 222 | | /* | | |
| 223 | | ioct->db_qid = 1; /* Queue ID number */ | | |
| 224 | | ioct->db_qsize = 80; /* Queue size */ | | |
| 225 | | ioct->db_inpt_ll = 80; /* Input line length */ | | |
| 226 | | ioct->db_histack = 10; /* History lines */ | | |
| 227 | | ioct->db_alias = 20; /* Number of alias */ | | |
| 228 | | ioct->db_symb = 20; /* Number of symbols */ | | |
| 229 | | ioct->db_symb = 2; /* Number of ports */ | | |
| 230 | | ioct->db_toggle = 0x1b; /* Toggle character */ | | |
| 231 | | ioct->db_xon = 0x1b; /* Xon */ | | |
| 232 | | ioct->db_xoff = 0x1b; /* Xoff */ | | |
| 233 | | ioct->db_exit = 0x1b; /* Exit character */ | | |
| 234 | | ioct->db_turdy = (long)Vrtx_tx; /* Hyperlink TERT driver addr */ | | |
| 235 | | ioct->db_in_filt = 0; /* Input filter */ | | |
| 236 | | ioct->db_out_filt = 0; /* Output filter */ | | |
| 237 | | ioct->db_pput1 = (long)rtscope_out_poll; /* Hyperlink input poll */ | | |
| 238 | | ioct->db_pput2 = 0; /* Hyperlink output poll */ | | |
| 239 | | ioct->db_pput2 = 0; /* Host input poll */ | | |
| 240 | | ioct->db_pput2 = 0; /* Host output poll */ | | |
| 241 | | ioct->db_spec_keys = (DB_SPEC *) 0; /* Special keys table */ | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM bsp.lm1000/bsp.c | DATE 5/23/89 | PAGE # 3/9 |
|--|--|---|-----------------|---------------|
| LINE # | | SOURCE TEXT | | |
| 221 | | /* and init_rtcable */ | | |
| 222 | | /* | | |
| 223 | | | | |
| 224 | | /* Create component configuration table | | |
| 225 | | | | |
| 226 | | /* | | |
| 227 | | init_cvt() | | |
| 228 | | { | | |
| 229 | | Cvt_Tbl *ct = (Cvt_Tbl *)(&(cftbl->cvt), | | |
| 230 | | short i, | | |
| 231 | | { | | |
| 232 | | ct->hr_max = HR_MAX, /* Ready system highest component no. */ | | |
| 233 | | ct->usr_max = USR_MAX, /* User highest component no. */ | | |
| 234 | | for(i = 0; i < 6; i++) /* Reserved */ | | |
| 235 | | ct->resl[i] = 0; | | |
| 236 | | for(i = 0; i < 8; i++) /* Reserved */ | | |
| 237 | | ct->resl[i] = 0; | | |
| 238 | | ct->comp_code = 0; /* No IOX */ | | |
| 239 | | ct->comp_work = 0; /* No IOX */ | | |
| 240 | | ct->comp_code = 0; /* No FDI */ | | |
| 241 | | ct->comp_work = 0; /* No FDI */ | | |
| 242 | | } | | |
| 243 | | /* and init_cvt */ | | |
| 244 | | /* | | |
| 245 | | | | |
| 246 | | /* Setup interrupt vector table | | |
| 247 | | | | |
| 248 | | /* | | |
| 249 | | setup_ivt() | | |
| 250 | | { | | |
| 251 | | unsigned long *ivt = (unsigned long *)VEC_BASE_ADDRESS, | | |
| 252 | | /* | | |
| 253 | | /* The bus handler | | |
| 254 | | /* | | |
| 255 | | *(ivt + VEC_BUS_ERROR) = (unsigned long) bus_isr, | | |
| 256 | | *(ivt + VEC_ADDRESS_ERROR) = (unsigned long) bus_isr, | | |
| 257 | | *(ivt + VEC_ILLEGAL_INSTRUCTION) = (unsigned long) bus_isr, | | |
| 258 | | /* | | |
| 259 | | /* Set up RTscope entry | | |
| 260 | | /* | | |
| 261 | | *(ivt + 0x21) = (RTS_BASE + RTS_ENTRY), /* RTscope entry point */ | | |
| 262 | | /* | | |
| 263 | | /* Set up Vrtx vectors | | |
| 264 | | /* | | |
| 265 | | *(ivt + 0x40) = (long)&(cftbl->v_cft), /* Config Table */ | | |
| 266 | | *(ivt + 0x40) = VRTX_BASE, /* VRTX Base */ | | |
| 267 | | /* | | |
| 268 | | /* Set up Serial port interrupt vectors | | |
| 269 | | /* | | |
| 270 | | /* To the ISR handler address to IVT Auto vector level 5 */ | | |
| 271 | | *(ivt + VEC_UART_INTERRUPT) = (long)serial_isr, | | |
| 272 | | /* | | |
| 273 | | /* Set up Counter/Timer interrupt vector | | |
| 274 | | /* | | |
| 275 | | /* Count/Timer ISR address to IVT 28, Auto vector level 4 */ | | |
| 276 | | *(ivt + VEC_TIMER_INTERRUPT) = (unsigned long)ct_isr, | | |
| 277 | | /* | | |
| 278 | | /* Set up ethernet interrupt vector | | |
| 279 | | /* | | |
| 280 | | /* ethernet ISR address to IVT 30, Auto vector level 6 */ | | |
| 281 | | *(ivt + VEC_LANCE_DOCKBELL_INTR) = (unsigned long)ether_isr, | | |
| 282 | | /* | | |
| 283 | | /* Clock board error ISR | | |
| 284 | | /* | | |
| 285 | | *(ivt + VEC_CB_INTERRUPT) = (unsigned long)error_isr, | | |
| 286 | | /* | | |
| 287 | | /* Parity ISR, IVT 31, MME | | |
| 288 | | /* | | |
| 289 | | *(ivt + VEC_PARITY_INTERRUPT) = (unsigned long)parity_isr, | | |
| 290 | | } | | |
| 291 | | /* end setup_ivt */ | | |
| 292 | | /* | | |
| 293 | | /* | | |
| 294 | | /* Enable cpu control register | | |
| 295 | | /* | | |
| 296 | | enable_cpu_reg() | | |
| 297 | | { | | |
| 298 | | cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG, | | |
| 299 | | /* | | |
| 300 | | /* enable timer gates, and enable the timer interrupt | | |
| 301 | | /* | | |
| 302 | | p_cpu_ctl_reg->timer_gate_0 = 1, | | |
| 303 | | p_cpu_ctl_reg->timer_gate_1 = 1, | | |
| 304 | | p_cpu_ctl_reg->timer_intr_ena_1 = 1, | | |
| 305 | | /* | | |
| 306 | | /* enable parity intr | | |
| 307 | | /* | | |
| 308 | | p_cpu_ctl_reg->parity_intr_ena = 1, | | |
| 309 | | /* | | |
| 310 | | /* enable interrupts | | |
| 311 | | /* | | |
| 312 | | p_cpu_ctl_reg->global_intr_ena = 1, | | |
| 313 | | } | | |
| 314 | | /* | | |
| 315 | | /* reset the CPU | | |
| 316 | | /* | | |
| 317 | | void | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM bsp.lm1000/bsp.c | DATE 5/23/89 | PAGE # 4/10 |
|--|---|------------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 361 | reset_cpu(reset_modeler_state) | | | |
| 362 | char reset_modeler_state; | | | |
| 363 | { | | | |
| 364 | char modeler_reset = reset_modeler_state; | | | |
| 365 | u_long err; | | | |
| 366 | cpu_control_reg_struct *cpu_ctl = (cpu_control_reg_struct *) CPU_CONTROL_REG; | | | |
| 367 | /* | | | |
| 368 | /* reason for reset | | | |
| 369 | */ | | | |
| 370 | if(reset_modeler_state != REBOOT) | | | |
| 371 | { | | | |
| 372 | (void)lm_svarm_access(&modeler_reset, MODELER_RESET, sizeof(MODELER_RESET), MEMORY_WRITE, (u_long *) &err); | | | |
| 373 | reset_modeler_state = SHUTDOWN; | | | |
| 374 | } | | | |
| 375 | (void)lm_svarm_access(&reset_modeler_state, MODELER_STATE, sizeof(MODELER_STATE), MEMORY_WRITE, (u_long *) &err); | | | |
| 376 | cpu_ctl->suicide = 1; | | | |
| 377 | for(;;) | | | |
| 378 | { | | | |
| 379 | } | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM | DATE | PAGE # |
|--|---|------------------------------|---------|------------|
| | | tasks.lm1000/hkeeping_task.c | 5/23/89 | 1/1 |
| | | | TIME | 4:42:33 pm |
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCCS_ID: hkeeping_task.c rev 3.1, 4/24/89 at 07:55:25 */ | | | |
| 2 | #include "common.h" | | | |
| 3 | #include "network.h" | | | |
| 4 | #include "cpu.h" | | | |
| 5 | #include "lance.h" | | | |
| 6 | #include "mod_err.h" | | | |
| 7 | #include "lm_rd_wr.h" | | | |
| 8 | #include "nvram.h" | | | |
| 9 | | | | |
| 10 | u_long calibrate_led = 0; | | | |
| 11 | u_long timer_semaphore; | | | |
| 12 | | | | |
| 13 | void clear_test_led(); | | | |
| 14 | void set_test_led(); | | | |
| 15 | u_short check_connection_for_life(); | | | |
| 16 | | | | |
| 17 | extern u_long lm_tick; | | | |
| 18 | extern u_long Got_a_char; | | | |
| 19 | extern u_long network_timeout; | | | |
| 20 | extern u_char lm_hardware_init_done; | | | |
| 21 | | | | |
| 22 | extern u_char fatal_hardware_error_encountered; | | | |
| 23 | extern u_char fatal_configuration_error_encountered; | | | |
| 24 | extern u_char non_fatal_configuration_error_encountered; | | | |
| 25 | | | | |
| 26 | extern BOOT_STRUCT boot; | | | |
| 27 | extern CONNECTION *table_of_conns(); | | | |
| 28 | | | | |
| 29 | #define MAX_TRIES 3 | | | |
| 30 | #define MSECSECONDS_PER_SECOND 1000 | | | |
| 31 | | | | |
| 32 | void | | | |
| 33 | housekeeping_task() | | | |
| 34 | { | | | |
| 35 | int err; | | | |
| 36 | u_char value = 0; | | | |
| 37 | u_char users = 0; | | | |
| 38 | extern BOOT_STRUCT boot; | | | |
| 39 | | | | |
| 40 | #ifdef BROKEN_HARDWARE | | | |
| 41 | extern char reinitialize_lance; | | | |
| 42 | | | | |
| 43 | #endif BROKEN_HARDWARE | | | |
| 44 | u_short short_network_timeout; | | | |
| 45 | register CONNECTION *conn_table; | | | |
| 46 | register CONNECTION *conn; | | | |
| 47 | | | | |
| 48 | extern char *lmel_version; | | | |
| 49 | | | | |
| 50 | | | | |
| 51 | modeler_state = MODELER_CALIBRATING; | | | |
| 52 | (void) lm_nvram_access(&modeler_state, MODELER_STATE, | | | |
| 53 | sizeof(MODELER_STATE), MEMORY_WRITE, (u_long *) &err); | | | |
| 54 | printf("\n%s\n", lmel_version); | | | |
| 55 | init(); | | | |
| 56 | read_hw_config(); | | | |
| 57 | init_mod_err(); | | | |
| 58 | enable_mod_err(); | | | |
| 59 | lm_hardware_init_done = TRUE; | | | |
| 60 | modeler_state = MODELER_RUNNING; | | | |
| 61 | (void) lm_nvram_access(&modeler_state, MODELER_STATE, | | | |
| 62 | sizeof(MODELER_STATE), MEMORY_WRITE, (u_long *) &err); | | | |
| 63 | | | | |
| 64 | if (fatal_hardware_error_encountered == TRUE) { | | | |
| 65 | printf("Fatal Hardware Error Encountered during start up.\n"); | | | |
| 66 | } | | | |
| 67 | if (fatal_configuration_error_encountered == TRUE) { | | | |
| 68 | printf("Fatal Configuration Error Encountered during start up.\n"); | | | |
| 69 | } | | | |
| 70 | if (non_fatal_configuration_error_encountered == TRUE) { | | | |
| 71 | printf("Non-Fatal Configuration Error Encountered during start up.\n"); | | | |
| 72 | } | | | |
| 73 | printf("Ready to accept network connections\n"); | | | |
| 74 | clear_test_led(); | | | |
| 75 | | | | |
| 76 | while (TRUE) { | | | |
| 77 | ac_send(timer_semaphore, 0, &err); | | | |
| 78 | if (err) { | | | |
| 79 | printf("\nError in send in housekeeping tx", err); | | | |
| 80 | } | | | |
| 81 | if (Got_a_char == 1) { | | | |
| 82 | Got_a_char = 0; | | | |
| 83 | printf("Running Core Modeler Code; this port is inactive.\n"); | | | |
| 84 | } | | | |
| 85 | #ifdef BROKEN_HARDWARE | | | |
| 86 | /* | | | |
| 87 | * if lance requires a jump start let's do it. | | | |
| 88 | */ | | | |
| 89 | if (reinitialize_lance == 1) { | | | |
| 90 | reinitialize_lance = 0; | | | |
| 91 | (void) ac_lock(); | | | |
| 92 | if (get_lance_ready_to_go() != SUCCESS) | | | |
| 93 | printf("Can't reinitialize lance\n"); | | | |
| 94 | else | | | |
| 95 | printf("Reinitialized lance\n"); | | | |
| 96 | if (start_lance() != SUCCESS) | | | |
| 97 | printf("Can't start lance\n"); | | | |
| 98 | else | | | |
| 99 | printf("Restarted lance\n"); | | | |
| 100 | (void) ac_unlock(); | | | |
| 101 | } | | | |
| 102 | #endif BROKEN_HARDWARE | | | |
| 103 | | | | |
| 104 | /* | | | |
| 105 | * read the boot structure. | | | |
| 106 | */ | | | |
| 107 | if (lm_nvram_access((char *) &boot, BOOT, sizeof(BOOT, | | | |
| 108 | MEMORY_READ, (u_long *) &err) == FAILURE) { | | | |
| 109 | printf("Failed to read boot structure.\n"); | | | |
| 110 | | | | |

| | | | | |
|--|--|------|------------|--------|
| Copyright 1989 Logic Modeling Systems | SOURCE PROGRAM tasks.lm1000/hkeeping_task.c | DATE | 5/23/89 | PAGE # |
| | | TIME | 4:42:33 pm | 2/2 |

| LINE # | SOURCE TEXT |
|--------|--|
| 121 | } |
| 122 | /* |
| 123 | set network timeout |
| 124 | */ |
| 125 | if (lm_syscall_access(&short_network_timeout, NETWORK_TIMEOUT, |
| 126 | sizeof NETWORK_TIMEOUT, MEMORY_READ, (u_long *) & net) == FAILURE) |
| 127 | printf("Unable to read network timeout\n"); |
| 128 | else { |
| 129 | |
| 130 | /* |
| 131 | Turn seconds to milliseconds |
| 132 | */ |
| 133 | network_timeout = short_network_timeout; |
| 134 | network_timeout *= MILLISECONDS_PER_SECOND; |
| 135 | } |
| 136 | |
| 137 | /* |
| 138 | Check reboot |
| 139 | */ |
| 140 | check_reboot(); |
| 141 | |
| 142 | /* |
| 143 | if we are calibrating DMC, set calibrate_led |
| 144 | */ |
| 145 | if (calibrate_led == 1) { |
| 146 | if (value == 0) { |
| 147 | clear_test_led(); |
| 148 | value = 1; |
| 149 | } else { |
| 150 | set_test_led(); |
| 151 | value = 0; |
| 152 | } |
| 153 | } |
| 154 | |
| 155 | /* |
| 156 | go thru conn structures checking for timeouts |
| 157 | */ |
| 158 | conn_table = table_of_conns(0); |
| 159 | for (users = 0; users != MAX_USERS; users++) { |
| 160 | conn = *conn_table++; |
| 161 | |
| 162 | if (conn == (CONNECTION *) NULL) |
| 163 | continue; |
| 164 | |
| 165 | CPU_DISABLE_INTERRUPTS; |
| 166 | |
| 167 | /* |
| 168 | close the connection |
| 169 | */ |
| 170 | if (conn->do_close == TRUE) { |
| 171 | if (close_connection_for_server(conn) == FAILURE) { |
| 172 | CPU_ENABLE_INTERRUPTS; |
| 173 | printf("Unable to close connection %d\n", conn->fd); |
| 174 | CPU_DISABLE_INTERRUPTS; |
| 175 | } |
| 176 | } |
| 177 | } else { |
| 178 | |
| 179 | /* |
| 180 | did we read the last part of the reply and is this |
| 181 | connection supposed to close |
| 182 | */ |
| 183 | if (conn->as_reading == FALSE && conn->as_closing == TRUE) { |
| 184 | conn->do_close = TRUE; |
| 185 | } |
| 186 | |
| 187 | /* |
| 188 | network timeout is set, and we are timing out |
| 189 | */ |
| 190 | |
| 191 | if (network_timeout != 0 && conn != (CONNECTION *) NULL && |
| 192 | conn->as_timing_out == TRUE) { |
| 193 | if (conn->time_to_live < (lm_tick * 5)) { |
| 194 | CPU_ENABLE_INTERRUPTS; |
| 195 | if (check_connection_for_life(conn) == FAILURE) { |
| 196 | printf("Can't check connection for life in housekeeping task user %d\n", |
| 197 | users); |
| 198 | } |
| 199 | conn->time_to_live = (lm_tick * 5) + network_timeout; |
| 200 | if (conn->number_of_live_retries > MAX_TRIES) { |
| 201 | |
| 202 | /* |
| 203 | The host device is not responding to device |
| 204 | requests. This does not mean that the host is dead, |
| 205 | so just make a note of it and continue. |
| 206 | */ |
| 207 | conn->number_of_live_retries = 0; |
| 208 | } |
| 209 | } |
| 210 | |
| 211 | if ((conn = table_of_conns(MAX_USERS)) == (CONNECTION *) NULL) { |
| 212 | CPU_ENABLE_INTERRUPTS; |
| 213 | continue; |
| 214 | } |
| 215 | |
| 216 | /* |
| 217 | close the connection |
| 218 | */ |
| 219 | if (conn->do_close == TRUE) { |
| 220 | if (close_connection_for_server(conn) == FAILURE) { |
| 221 | CPU_ENABLE_INTERRUPTS; |
| 222 | printf("Unable to close connection %d\n", conn->fd); |
| 223 | } |
| 224 | CPU_DISABLE_INTERRUPTS; |
| 225 | continue; |
| 226 | } |
| 227 | |
| 228 | /* |
| 229 | Did we send the 1 packet reply and is this connection supposed to |
| 230 | close? |
| 231 | */ |
| 232 | if (conn->as_closing == TRUE) { |
| 233 | conn->do_close = TRUE; |
| 234 | } |
| 235 | CPU_ENABLE_INTERRUPTS; |
| 236 | |
| 237 | } |
| 238 | |
| 239 | |
| 240 | void |

1731

5,353,243

1732

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM | DATE | PAGE # |
|--|---|------------------------------|------------|--------|
| | | tasks.lm1000/hkeeping_task.c | 5/23/89 | |
| | | | TIME | 3/3 |
| | | | 4:42:33 pm | |
| LINE # | SOURCE TEXT | | | |
| 241 | set_test_led() | | | |
| 242 | { | | | |
| 243 | cpu_control_reg_struct *p_cpu_ctl_reg = | | | |
| 244 | (cpu_control_reg_struct *) CPU_CONTROL_REG, | | | |
| 245 | { | | | |
| 246 | p_cpu_ctl_reg->not_test_led = not_LED_ON, | | | |
| 247 | } | | | |
| 248 | void | | | |
| 249 | clear_test_led() | | | |
| 250 | { | | | |
| 251 | { | | | |
| 252 | cpu_control_reg_struct *p_cpu_ctl_reg = | | | |
| 253 | (cpu_control_reg_struct *) CPU_CONTROL_REG, | | | |
| 254 | { | | | |
| 255 | p_cpu_ctl_reg->not_test_led = not_LED_OFF, | | | |
| 256 | } | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM | DATE | PAGE # |
|--|--|-----------------------------|-----------------|--------|
| | | tasks.lm1000/receive_task.c | 5/23/89 | |
| | | | TIME 4:42:34 pm | 1/4 |
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCCS ID: receive_task.c rev 1.1, 4/24/89 at 07:55:23 */ | | | |
| 2 | /* | | | |
| 3 | ** receive_task.c | | | |
| 4 | */ | | | |
| 5 | #include "common.h" | | | |
| 6 | #include "fifo.h" | | | |
| 7 | #include "cpu.h" | | | |
| 8 | #include "task.h" | | | |
| 9 | #include "message.h" | | | |
| 10 | #include "network.h" | | | |
| 11 | | | | |
| 12 | u_long lm_global_clock; | | | |
| 13 | | | | |
| 14 | extern CONNECTION *table_of_conns[1]; | | | |
| 15 | | | | |
| 16 | void | | | |
| 17 | receive_task() | | | |
| 18 | { | | | |
| 19 | u_short type; | | | |
| 20 | char str[MAX_MESSAGE]; | | | |
| 21 | | | | |
| 22 | u_long rta; | | | |
| 23 | u_char user; | | | |
| 24 | | | | |
| 25 | CONNECTION *conn; | | | |
| 26 | | | | |
| 27 | /* | | | |
| 28 | ** Initialize socket; before doing anything else | | | |
| 29 | ** sets up base conn. | | | |
| 30 | */ | | | |
| 31 | if (init_socket() == FAILURE) { | | | |
| 32 | /* EXIT */ | | | |
| 33 | } | | | |
| 34 | | | | |
| 35 | lm_global_clock = 0; | | | |
| 36 | for (;;) { | | | |
| 37 | { | | | |
| 38 | rta = lm_choose_connection(user); | | | |
| 39 | if (rta != FAILURE) | | | |
| 40 | { | | | |
| 41 | conn = table_of_conns[user]; | | | |
| 42 | if (rta == PENDING) | | | |
| 43 | { | | | |
| 44 | if (lm_send_reply(conn) != SUCCESS) | | | |
| 45 | { | | | |
| 46 | printf("aaaaaaaaaaaaaa"), | | | |
| 47 | /* EXIT */ | | | |
| 48 | } | | | |
| 49 | } | | | |
| 50 | else | | | |
| 51 | { | | | |
| 52 | /* EXIT */ | | | |
| 53 | } | | | |
| 54 | } | | | |
| 55 | else | | | |
| 56 | { | | | |
| 57 | while (lm_dequeue_message(&type, str) != FAILURE) { | | | |
| 58 | if (type == ERROR_MSG) | | | |
| 59 | (void) printf("ERROR: %s\n", str); | | | |
| 60 | else (void) printf("WARNING: %s\n", str); | | | |
| 61 | } | | | |
| 62 | } | | | |
| 63 | } | | | |
| 64 | } | | | |
| 65 | } | | | |
| 66 | } | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/arp.c

DATE 5/23/89
TIME 4:42:10 pm

PAGE #
1/1

```

1  /* SCCS ID: arp.c rev 3.1, 4/24/89 at 07:46:29 */
2
3  /*
4   * This file contains the ARP request and response
5   * functions. For information on ARP and RARP see
6   * the DDM protocol handbook, volume 3, pp 3-615.
7   */
8
9  #include "common.h"
10 #include "arp.h"
11
12 /* external function declarations */
13 extern unsigned short lance_transmit();
14 extern char *memcpy();
15
16 /*
17  * This routine broadcasts an ARP request for a modeler.
18  * The receive task will receive the reply and handle
19  * appropriately
20  */
21
22 void send_arp_request (enet_address, inet_address, dest_inet_address)
23 {
24     char *enet_address;
25     unsigned long inet_address;
26     unsigned long dest_inet_address;
27
28     static struct ethernet_arp_packet arp_request;
29     long error;
30
31     (void) memcpy ((char *) arp_request.destination, BROADCAST, ARP_HARDWARE_SIZE);
32     (void) memcpy ((char *) arp_request.source, enet_address, ARP_HARDWARE_SIZE);
33
34     arp_request.type = ARP_ARP;
35
36     arp_request.arp.arp_hardware_type = ARP_ETHERNET;
37     arp_request.arp.arp_protocol_type = ARP_IP;
38     arp_request.arp.arp_hardware_address_length = ARP_HARDWARE_SIZE;
39     arp_request.arp.arp_protocol_address_length = ARP_PROTOCOL_SIZE;
40     arp_request.arp.arp_opcode = ARP_REQUEST;
41     arp_request.arp.arp_source_protocol_address = inet_address;
42     arp_request.arp.arp_target_protocol_address = dest_inet_address;
43
44     (void) memcpy ((char *) arp_request.arp.arp_source_hardware_address, enet_address, ARP_HARDWARE_SIZE);
45     (void) memcpy ((char *) arp_request.arp.arp_target_hardware_address, NULL_ADDRESS, ARP_HARDWARE_SIZE);
46
47     (void) lance_transmit (&arp_request, (short) sizeof (arp_request), &error);
48
49     return;
50 }
51
52
53
54
55 /* This routine generates a response to an ARP request. It is
56  * currently unimplemented. The modeler will not respond to
57  * ARP requests, a host machine must publish an ARP response
58  * for a modeler. I wrote this routine simply because it was
59  * easy to do while I had my name in the DDM book. If we ever
60  * implement a full ARP, I hope I saved someone some time and
61  * effort, if not, oh well...
62  */
63
64 void send_arp_reply (arp_reply, enet_address, inet_address, dest_enet_address, dest_inet_address)
65 {
66     struct ethernet_arp_packet *arp_reply;
67     char *enet_address;
68     unsigned long inet_address;
69     char *dest_enet_address;
70     unsigned long dest_inet_address;
71
72     long error;
73
74     (void) memcpy ((char *) arp_reply->destination, dest_enet_address, ARP_HARDWARE_SIZE);
75     (void) memcpy ((char *) arp_reply->source, enet_address, ARP_HARDWARE_SIZE);
76
77     arp_reply->type = ARP_ARP;
78
79     arp_reply->arp.arp_hardware_type = ARP_ETHERNET;
80     arp_reply->arp.arp_protocol_type = ARP_IP;
81     arp_reply->arp.arp_hardware_address_length = ARP_HARDWARE_SIZE;
82     arp_reply->arp.arp_protocol_address_length = ARP_PROTOCOL_SIZE;
83     arp_reply->arp.arp_opcode = ARP_REPLY;
84     arp_reply->arp.arp_source_protocol_address = inet_address;
85     arp_reply->arp.arp_target_protocol_address = dest_inet_address;
86
87     (void) memcpy ((char *) arp_reply->arp.arp_source_hardware_address, enet_address, ARP_HARDWARE_SIZE);
88     (void) memcpy ((char *) arp_reply->arp.arp_target_hardware_address, dest_enet_address, ARP_HARDWARE_SIZE);
89
90     (void) lance_transmit (arp_reply, (short) sizeof (arp_reply), &error);
91
92     return;
93 }

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/bus.c | DATE 5/23/89 TIME 4:42:10 pm | PAGE # 1/2 |
|--|--|----------------------------|---------------------------------------|---------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCCS ID: bus.c rev 3.1, 4/24/89 at 07:46:32 */ | | | |
| 2 | /* | | | |
| 3 | ** bus.c | | | |
| 4 | ** bus error handler. | | | |
| 5 | ** It is more complicated than it needs to be, | | | |
| 6 | ** we need to use this to determine if a PAC | | | |
| 7 | ** is present... | | | |
| 8 | */ | | | |
| 9 | #include "common.h" | | | |
| 10 | #include "cpu.h" | | | |
| 11 | #include "cpu_vec.h" | | | |
| 12 | #include "mod_err.h" | | | |
| 13 | #include "nvaram.h" | | | |
| 14 | #include "lm_rd_wr.h" | | | |
| 15 | #include "bus.h" | | | |
| 16 | static char buf[200]; | | | |
| 17 | extern bus_isr(); | | | |
| 18 | u_long setup_ivt_bus(); | | | |
| 19 | extern u_short lm_nvaram_access(); | | | |
| 20 | extern void reset_cpu(); | | | |
| 21 | void bus_error_tx(); | | | |
| 22 | void output_routine(); | | | |
| 23 | u_short address_in_map(); | | | |
| 24 | static u_char *ignore_low_address; | | | |
| 25 | static u_char *ignore_hi_address; | | | |
| 26 | u_char ignore_mem_error; | | | |
| 27 | static u_char mem_error; | | | |
| 28 | static char debug_bus = 1; | | | |
| 29 | /* | | | |
| 30 | ** Bus error handler | | | |
| 31 | */ | | | |
| 32 | void bus_error() | | | |
| 33 | { | | | |
| 34 | extern STACK_FRAME *bus_error_address; | | | |
| 35 | STACK_FRAME *stack_frame = bus_error_address; | | | |
| 36 | u_long err; | | | |
| 37 | if(address_in_map(stack_frame->address) == SUCCESS) | | | |
| 38 | { | | | |
| 39 | if(ignore_mem_error == TRUE) | | | |
| 40 | { | | | |
| 41 | if(ignore_low_address >= stack_frame->address && | | | |
| 42 | ignore_hi_address <= stack_frame->address) | | | |
| 43 | { | | | |
| 44 | if(stack_frame->spec_status_word & 0x0100) | | | |
| 45 | { | | | |
| 46 | stack_frame->spec_status_word = 0x0100; | | | |
| 47 | mem_error = TRUE; | | | |
| 48 | return; | | | |
| 49 | } | | | |
| 50 | } | | | |
| 51 | } | | | |
| 52 | if(lm_nvaram_access((char *) stack_frame->address, BUS_ADDR, (u_long) sizeof BUS_ADDR, MEMORY_WRITE, &err) == FAILURE) | | | |
| 53 | { | | | |
| 54 | reset_cpu(SUICIDE); | | | |
| 55 | } | | | |
| 56 | if(lm_nvaram_access((char *) stack_frame->spec_status_word, SSW_REG, (u_long) sizeof SSW_REG, MEMORY_WRITE, &err) == FAILURE) | | | |
| 57 | { | | | |
| 58 | reset_cpu(SUICIDE); | | | |
| 59 | } | | | |
| 60 | if(lm_nvaram_access((char *) stack_frame->vector, ERROR_VECTOR, (u_long) sizeof ERROR_VECTOR, MEMORY_WRITE, &err) == FAILURE) | | | |
| 61 | { | | | |
| 62 | reset_cpu(SUICIDE); | | | |
| 63 | } | | | |
| 64 | /* | | | |
| 65 | ** print out screen | | | |
| 66 | */ | | | |
| 67 | if(debug_bus == 1) | | | |
| 68 | { | | | |
| 69 | sprintf(buf, "Interrupt priority level %0x", (stack_frame->sr & 0x700) >> 8); | | | |
| 70 | output_routine(buf); | | | |
| 71 | sprintf(buf, "Program Counter %0x", stack_frame->pc); | | | |
| 72 | output_routine(buf); | | | |
| 73 | sprintf(buf, "Failed at address %0x", stack_frame->address); | | | |
| 74 | output_routine(buf); | | | |
| 75 | if(stack_frame->spec_status_word & 0x100) | | | |
| 76 | { | | | |
| 77 | sprintf(buf, "Data Access"); | | | |
| 78 | output_routine(buf); | | | |
| 79 | if(stack_frame->spec_status_word & 0x80) | | | |
| 80 | { | | | |
| 81 | sprintf(buf, "Read Modify write cycle "); | | | |
| 82 | output_routine(buf); | | | |
| 83 | } | | | |
| 84 | if(stack_frame->spec_status_word & 0x10) | | | |
| 85 | { | | | |
| 86 | sprintf(buf, "Byte "); | | | |
| 87 | output_routine(buf); | | | |
| 88 | } | | | |
| 89 | if(stack_frame->spec_status_word & 0x20) | | | |
| 90 | { | | | |
| 91 | sprintf(buf, "Word "); | | | |
| 92 | output_routine(buf); | | | |
| 93 | } | | | |
| 94 | if(((stack_frame->spec_status_word & 0x30) | | | |
| 95 | { | | | |
| 96 | sprintf(buf, "Long "); | | | |
| 97 | output_routine(buf); | | | |
| 98 | } | | | |
| 99 | if(stack_frame->spec_status_word & 0x40) | | | |
| 100 | { | | | |
| 101 | sprintf(buf, "Read cycle "); | | | |
| 102 | output_routine(buf); | | | |
| 103 | } | | | |
| 104 | else | | | |
| 105 | { | | | |
| 106 | sprintf(buf, "Write cycle "); | | | |
| 107 | output_routine(buf); | | | |
| 108 | } | | | |
| 109 | } | | | |
| 110 | } | | | |
| 111 | if(stack_frame->spec_status_word & 0x1000) | | | |
| 112 | { | | | |
| 113 | sprintf(buf, "Instruction Access "); | | | |
| 114 | output_routine(buf); | | | |
| 115 | } | | | |
| 116 | /* | | | |
| 117 | ** we crashed | | | |
| 118 | */ | | | |
| 119 | if(address_in_map(stack_frame->address) == FAILURE) | | | |
| 120 | { | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/bus.c | DATE 5/23/89 | PAGE # 2/3 |
|--|--|----------------------------|-----------------|---------------|
| LINE # | SOURCE TEXT | | | |
| 121 | reset_cpu(BUS_ERROR); | | | |
| 122 | } | | | |
| 123 | } | | | |
| 124 | #define ALWAYS_MEMORY 0 | | | |
| 125 | #define NEVER_MEMORY 1 | | | |
| 126 | #define OPTIONAL_MEMORY 2 | | | |
| 127 | #define MAP_SIZE 32 | | | |
| 128 | static u_long map[MAP_SIZE][2] = { | | | |
| 129 | 0x00000000, ALWAYS_MEMORY, | | | |
| 130 | 0x00000000, OPTIONAL_MEMORY, | | | |
| 131 | 0x10000000, NEVER_MEMORY, | | | |
| 132 | 0x10000000, OPTIONAL_MEMORY, | | | |
| 133 | 0x10000000, ALWAYS_MEMORY, | | | |
| 134 | 0x10000000, NEVER_MEMORY, | | | |
| 135 | 0x20000000, NEVER_MEMORY, | | | |
| 136 | 0x20000000, OPTIONAL_MEMORY, | | | |
| 137 | 0x20000000, NEVER_MEMORY, | | | |
| 138 | 0x20000000, OPTIONAL_MEMORY, | | | |
| 139 | 0x30000000, NEVER_MEMORY, | | | |
| 140 | 0x30000000, OPTIONAL_MEMORY, | | | |
| 141 | 0x30000000, NEVER_MEMORY, | | | |
| 142 | 0x30000000, OPTIONAL_MEMORY, | | | |
| 143 | 0x40000000, NEVER_MEMORY, | | | |
| 144 | 0x40000000, OPTIONAL_MEMORY, | | | |
| 145 | 0x40000000, NEVER_MEMORY, | | | |
| 146 | 0x40000000, OPTIONAL_MEMORY, | | | |
| 147 | 0x50000000, NEVER_MEMORY, | | | |
| 148 | 0x50000000, OPTIONAL_MEMORY, | | | |
| 149 | 0x50000000, NEVER_MEMORY, | | | |
| 150 | 0x50000000, OPTIONAL_MEMORY, | | | |
| 151 | 0x60000000, NEVER_MEMORY, | | | |
| 152 | 0x60000000, OPTIONAL_MEMORY, | | | |
| 153 | 0x60000000, NEVER_MEMORY, | | | |
| 154 | 0x60000000, OPTIONAL_MEMORY, | | | |
| 155 | 0x70000000, NEVER_MEMORY, | | | |
| 156 | 0x70000000, OPTIONAL_MEMORY, | | | |
| 157 | 0x70000000, NEVER_MEMORY, | | | |
| 158 | 0x70000000, OPTIONAL_MEMORY, | | | |
| 159 | 0x80000000, NEVER_MEMORY, | | | |
| 160 | 0x80000000, NEVER_MEMORY, | | | |
| 161 | }; | | | |
| 162 | u_short | | | |
| 163 | address_in_map(address) | | | |
| 164 | u_char *address; | | | |
| 165 | { | | | |
| 166 | register u_char i; | | | |
| 167 | /* | | | |
| 168 | ** Go thru above table to determine what to do.. | | | |
| 169 | */ | | | |
| 170 | for(i = 0; i <= MAP_SIZE; ++i) | | | |
| 171 | { | | | |
| 172 | if(map[i][0] >= (u_long) address) | | | |
| 173 | { | | | |
| 174 | if(map[i][1] == NEVER_MEMORY) | | | |
| 175 | return(FAILURE); | | | |
| 176 | return(SUCCESS); | | | |
| 177 | } | | | |
| 178 | } | | | |
| 179 | return(FAILURE); | | | |
| 180 | } | | | |
| 181 | /* | | | |
| 182 | ** address for which errors are to be ignored | | | |
| 183 | */ | | | |
| 184 | void lm_ignore_bus_error(addr_lo, addr_hi) | | | |
| 185 | u_char *addr_lo, *addr_hi; | | | |
| 186 | { | | | |
| 187 | ignore_low_address = addr_lo; | | | |
| 188 | ignore_hi_address = addr_hi; | | | |
| 189 | ignore_mem_error = TRUE; | | | |
| 190 | mem_error = FALSE; | | | |
| 191 | } | | | |
| 192 | /* | | | |
| 193 | ** did we get a bus error | | | |
| 194 | */ | | | |
| 195 | lm_check_bus_error() | | | |
| 196 | { | | | |
| 197 | return(mem_error); | | | |
| 198 | } | | | |
| 199 | /* | | | |
| 200 | ** deal with bus error correctly, and do not ignore any errors | | | |
| 201 | */ | | | |
| 202 | void lm_enable_bus_error() | | | |
| 203 | { | | | |
| 204 | ignore_mem_error = FALSE; | | | |
| 205 | } | | | |
| 206 | /* | | | |
| 207 | ** read | | | |
| 208 | */ | | | |
| 209 | u_short | | | |
| 210 | lm_read_probe(addr) | | | |
| 211 | register u_long *addr; | | | |
| 212 | { | | | |
| 213 | cpu_control_reg_struct *cpu_control_register = (cpu_control_reg_struct *) CPU_CONTROL_REG; | | | |
| 214 | u_long junk; | | | |
| 215 | temp = setup_ivt_bus((u_long) bus_iar); | | | |
| 216 | lm_ignore_bus_error((u_char *) addr, (u_char *) addr); | | | |
| 217 | /* | | | |
| 218 | ** read | | | |
| 219 | */ | | | |
| 220 | junk = *addr; | | | |
| 221 | lm_enable_bus_error(); | | | |
| 222 | (void) setup_ivt_bus(temp); | | | |
| 223 | if(lm_check_bus_error() == TRUE) | | | |
| 224 | return(FAILURE); | | | |
| 225 | return(SUCCESS); | | | |
| 226 | } | | | |
| 227 | /* | | | |
| 228 | ** write | | | |
| 229 | */ | | | |
| 230 | u_short | | | |
| 231 | lm_write_probe(addr, value) | | | |
| 232 | register u_long *addr, value; | | | |
| 233 | { | | | |
| 234 | u_long temp; | | | |
| 235 | cpu_control_reg_struct *cpu_control_register = (cpu_control_reg_struct *) CPU_CONTROL_REG; | | | |
| 236 | temp = setup_ivt_bus((u_long) bus_iar); | | | |
| 237 | lm_ignore_bus_error((u_char *) addr, (u_char *) addr); | | | |
| 238 | /* | | | |
| 239 | ** write | | | |
| 240 | */ | | | |
| 241 | *addr = value; | | | |
| 242 | lm_enable_bus_error(); | | | |

1741

5,353,243

1742

Copyright 1989
Logic Modeling SystemsSOURCE PROGRAM
os/bus.cDATE 5/23/89
TIME 4:42:10 pmPAGE #
3/4

| LINE # | SOURCE TEXT |
|--------|---|
| 241 | (void) setup_ivt_bus(temp); |
| 242 | if(! check_bus_error() == TRUE) |
| 243 | return(FAILURE); |
| 244 | return(SUCCESS); |
| 245 | } |
| 246 | static |
| 247 | u_long setup_ivt_bus(bus_iar) |
| 248 | u_long bus_iar; |
| 249 | { |
| 250 | unsigned long *ivt = (unsigned long *)VEC_BASE_ADDRESS; |
| 251 | u_long temp; |
| 252 | /* |
| 253 | /* The bus handler |
| 254 | */ |
| 255 | temp = *(ivt + 2); |
| 256 | *(ivt + 2) = (unsigned long) bus_iar; |
| 257 | return(temp); |
| 258 | } |
| 259 | |
| 260 | void output_routine(buf) |
| 261 | char *buf; |
| 262 | { |
| 263 | char *bufptr = buf; |
| 264 | while(*bufptr) |
| 265 | { |
| 266 | Bus_error_tx(*bufptr++); |
| 267 | } |
| 268 | } |
| 269 | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/duart.c

DATE 5/23/89
TIME 4:42:10 pm

PAGE #
1/5

```

1  /* SCSS_ID: duart.c Rev 3.1, 4/24/89 at 07:46:36 */
2  /*
3  /* Duart.c
4  /*
5  /*
6  #include "cpu.h"
7  #include "duart.h"
8  /*
9  /* have UARTA & UARTB been initialized
10 /* 0 = no, 1 = yes.
11 /*
12 extern INTERRUPT_REG Wart_mask;
13 /*
14 int lm_uarta_baud; /* current uarta b baud rate */
15 int lm_uartb_baud; /* current uartb b baud rate */
16 /*
17 /* Varta_init()
18 /* Initialize channel A of the DUART.
19 /*
20 Varta_init( baud )
21 unsigned int baud;
22 {
23     MODE_REG_1 mode_reg1a = (MODE_REG_1 *) (CPU_DUART + MODE_REG1_A);
24     MODE_REG_2 mode_reg2a = (MODE_REG_2 *) (CPU_DUART + MODE_REG2_A);
25     CMD_REG cmd_rega = (CMD_REG *) (CPU_DUART + CMD_REG_A);
26     ACR *p_acr = (ACR *) (CPU_DUART + AUX_CNTL_REG);
27     /*
28     /* reset everything
29     /* through command register
30     /*
31     /* disable tx & rx
32     /*
33     cmd_rega.zero = 0;
34     cmd_rega.command = NOP;
35     cmd_rega.dis_tx = DUART_DISABLE;
36     cmd_rega.enb_tx = DUART_DISABLE;
37     cmd_rega.dis_rx = DUART_DISABLE;
38     cmd_rega.enb_rx = DUART_DISABLE;
39     *p_cmd_rega = cmd_rega;
40     /*
41     /* reset channel
42     /*
43     cmd_rega.command = RESET_CH;
44     *p_cmd_rega = cmd_rega;
45     /*
46     /* reset error
47     /*
48     cmd_rega.command = RESET_ERR;
49     *p_cmd_rega = cmd_rega;
50     /*
51     /* reset rcvr
52     /*
53     cmd_rega.command = RESET_RCVR;
54     *p_cmd_rega = cmd_rega;
55     /*
56     /* reset tx
57     /*
58     cmd_rega.command = RESET_TX;
59     *p_cmd_rega = cmd_rega;
60     /*
61     /* reset mode register
62     /*
63     cmd_rega.command = RESET_MODE_REG;
64     *p_cmd_rega = cmd_rega;
65     /*
66     /* set up mode reg 1
67     /* char mode, 8 bits/char & no parity
68     /*
69     mode_reg1a.tx_rts = DUART_DISABLE;
70     mode_reg1a.rx_int = RX_INT_RDY;
71     mode_reg1a.error_mode = ERR_MODE_CHAR;
72     mode_reg1a.parity = PARITY_NONE;
73     mode_reg1a.bits_per_char = BITS_PER_CHAR_8;
74     *p_mode_reg1a = mode_reg1a;
75     /*
76     /* set up mode reg 2
77     /*
78     mode_reg2a.chas_mode = CH_MODE_NORMAL;
79     mode_reg2a.tx_rts = DUART_DISABLE;
80     mode_reg2a.cts_enable = DUART_DISABLE;
81     mode_reg2a.stop_les = STOP_BITS_2;
82     *p_mode_reg2a = mode_reg2a;
83     /*
84     /* clock select
85     /*
86     Varta_baud( baud );
87     /*
88     /* set up aux. control reg
89     /*
90     acr.zero = 0;
91     *p_acr = acr;
92     /*
93     /* set up int. mask reg
94     /* this controls ints. for both UART's
95     /* we must take this into consideration.
96     /*
97     Varta_inr();
98     /*
99     /* enable tx & rx operations
100    /*
101    cmd_rega.command = NOP;
102    cmd_rega.dis_tx = DUART_DISABLE;
103    cmd_rega.enb_tx = DUART_ENABLE;
104    cmd_rega.dis_rx = DUART_DISABLE;
105    cmd_rega.enb_rx = DUART_ENABLE;
106    *p_cmd_rega = cmd_rega;
107    }
108    Varta_baud( baud )
109    {
110        CLK_SEL clk_sel; *p_clk_sel = (CLK_SEL *) (CPU_DUART + CLK_SEL_REG_A);
111        lm_uarta_baud = baud & 0xf;
112    }

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/duart.c | DATE 5/23/89 TIME 4:42:10 pm | PAGE # 2/6 |
|--|--|------------------------------|---------------------------------|---------------|
| LINE # | SOURCE TEXT | | | |
| 121 | /* | | | |
| 122 | ** clock select | | | |
| 123 | */ | | | |
| 124 | clk_sela.RX_clk = baud; | | | |
| 125 | clk_sela.TX_clk = baud; | | | |
| 126 | *p_clk_sela = clk_sela; | | | |
| 127 | } | | | |
| 128 | | | | |
| 129 | uartb_irq() | | | |
| 130 | { | | | |
| 131 | INTERRUPT_REG interrupt_reg, *p_interrupt_reg = (INTERRUPT_REG *) (CPU_DUART + INT_MASK_REG); | | | |
| 132 | /* | | | |
| 133 | ** set up int. mask reg | | | |
| 134 | ** this controls ints. for both UART's | | | |
| 135 | ** we must take this into consideration. | | | |
| 136 | */ | | | |
| 137 | interrupt_reg = uartb_mask; | | | |
| 138 | interrupt_reg.in_port = DUART_DISABLE; | | | |
| 139 | interrupt_reg.delta_brk_b = DUART_DISABLE; | | | |
| 140 | interrupt_reg.counter = DUART_DISABLE; | | | |
| 141 | interrupt_reg.delta_brk_a = DUART_DISABLE; | | | |
| 142 | interrupt_reg.rx_rdy_a = DUART_ENABLE; | | | |
| 143 | interrupt_reg.tx_rdy_a = DUART_ENABLE; | | | |
| 144 | /* | | | |
| 145 | ** set interrupt reg. and save in mask this is a write only reg. | | | |
| 146 | */ | | | |
| 147 | uartb_mask = interrupt_reg; | | | |
| 148 | *p_interrupt_reg = interrupt_reg; | | | |
| 149 | } | | | |
| 150 | | | | |
| 151 | | | | |
| 152 | | | | |
| 153 | uartb_init(baud) | | | |
| 154 | { | | | |
| 155 | unassigned int baud; | | | |
| 156 | /* | | | |
| 157 | MODE_REG_1 mode_reg1b, *p_mode_reg1b = (MODE_REG_1 *) (CPU_DUART + MODE_REG1_B); | | | |
| 158 | MODE_REG_2 mode_reg2b, *p_mode_reg2b = (MODE_REG_2 *) (CPU_DUART + MODE_REG2_B); | | | |
| 159 | CLK_SEL clk_selb, *p_clk_selb = (CLK_SEL *) (CPU_DUART + CLK_SEL_REG_B); | | | |
| 160 | CMD_REG cmd_regb, *p_cmd_regb = (CMD_REG *) (CPU_DUART + CMD_REG_B); | | | |
| 161 | /* | | | |
| 162 | in_uartb_baud = baud & 0xf; | | | |
| 163 | /* | | | |
| 164 | ** reset everything | | | |
| 165 | ** through command register | | | |
| 166 | /* | | | |
| 167 | ** disable tx & rx | | | |
| 168 | /* | | | |
| 169 | cmd_regb.iaro = 0; | | | |
| 170 | cmd_regb.command = NOP; | | | |
| 171 | cmd_regb.dis_tx = DUART_ENABLE; | | | |
| 172 | cmd_regb.ena_tx = DUART_DISABLE; | | | |
| 173 | cmd_regb.dis_rx = DUART_ENABLE; | | | |
| 174 | cmd_regb.ena_rx = DUART_DISABLE; | | | |
| 175 | *p_cmd_regb = cmd_regb; | | | |
| 176 | /* | | | |
| 177 | ** reset channel | | | |
| 178 | /* | | | |
| 179 | cmd_regb.command = RESET_CH; | | | |
| 180 | *p_cmd_regb = cmd_regb; | | | |
| 181 | /* | | | |
| 182 | ** reset error | | | |
| 183 | /* | | | |
| 184 | cmd_regb.command = RESET_ERR; | | | |
| 185 | *p_cmd_regb = cmd_regb; | | | |
| 186 | /* | | | |
| 187 | ** reset rxvr | | | |
| 188 | /* | | | |
| 189 | cmd_regb.command = RESET_RXVR; | | | |
| 190 | *p_cmd_regb = cmd_regb; | | | |
| 191 | /* | | | |
| 192 | ** reset tx | | | |
| 193 | /* | | | |
| 194 | cmd_regb.command = RESET_TX; | | | |
| 195 | *p_cmd_regb = cmd_regb; | | | |
| 196 | /* | | | |
| 197 | ** reset mode register | | | |
| 198 | /* | | | |
| 199 | cmd_regb.command = RESET_MODE_REG; | | | |
| 200 | *p_cmd_regb = cmd_regb; | | | |
| 201 | /* | | | |
| 202 | ** set up mode reg 1 | | | |
| 203 | ** char mode, 8-bits/char & no parity | | | |
| 204 | /* | | | |
| 205 | mode_reg1b.rx_rts = DUART_DISABLE; | | | |
| 206 | mode_reg1b.rx_int = RX_INT_RDY; | | | |
| 207 | mode_reg1b.error_mode = ERR_MODE_CHAR; | | | |
| 208 | mode_reg1b.parity = PARITY_NONE; | | | |
| 209 | mode_reg1b.bits_per_char = BITS_PER_CHAR_8; | | | |
| 210 | *p_mode_reg1b = mode_reg1b; | | | |
| 211 | /* | | | |
| 212 | ** set up mode reg 2 | | | |
| 213 | /* | | | |
| 214 | mode_reg2b.as_mode = CE_MODE_NORMAL; | | | |
| 215 | mode_reg2b.rx_rts = DUART_DISABLE; | | | |
| 216 | mode_reg2b.cts_enable = DUART_DISABLE; | | | |
| 217 | mode_reg2b.stop_lee = STOP_BITS_2; | | | |
| 218 | *p_mode_reg2b = mode_reg2b; | | | |
| 219 | /* | | | |
| 220 | ** clock select | | | |
| 221 | /* | | | |
| 222 | uartb_baud(baud); | | | |
| 223 | /* | | | |
| 224 | ** set up int. mask reg | | | |
| 225 | ** this controls ints. for both UART's | | | |
| 226 | ** we must take this into consideration. | | | |
| 227 | /* | | | |
| 228 | uartb_irq(); | | | |
| 229 | /* | | | |
| 230 | ** enable tx & rx operations | | | |
| 231 | /* | | | |
| 232 | cmd_regb.command = NOP; | | | |
| 233 | cmd_regb.dis_tx = DUART_DISABLE; | | | |
| 234 | cmd_regb.ena_tx = DUART_ENABLE; | | | |
| 235 | cmd_regb.dis_rx = DUART_DISABLE; | | | |
| 236 | cmd_regb.ena_rx = DUART_ENABLE; | | | |
| 237 | *p_cmd_regb = cmd_regb; | | | |
| 238 | /* | | | |
| 239 | | | | |
| 240 | | | | |

1747

5,353,243

1748

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/duart.c | DATE 5/23/89 | PAGE # 3/7 |
|--|---|------------------------------|-----------------|---------------|
| LINE # | SOURCE TEXT | | | |
| 241 | } | | | |
| 242 | uart_baud(baud) | | | |
| 243 | unsigned int baud; | | | |
| 244 | { | | | |
| 245 | CLK_SEL clk_selb, *p_clk_selb = (CLK_SEL *) (CPU_DUART + CLK_SEL_REG_B), | | | |
| 246 | lm_uartb_baud = baud & 0xf, | | | |
| 247 | /* | | | |
| 248 | ** clock select | | | |
| 249 | */ | | | |
| 250 | clk_selb.RX_clk = baud; | | | |
| 251 | clk_selb.TX_clk = baud; | | | |
| 252 | *p_clk_selb = clk_selb; | | | |
| 253 | } | | | |
| 254 | uart_isr() | | | |
| 255 | { | | | |
| 256 | INTERRUPT_REG interrupt_reg, *p_interrupt_reg = (INTERRUPT_REG *) (CPU_DUART + INT_MASK_REG), | | | |
| 257 | /* | | | |
| 258 | ** set up int. mask reg | | | |
| 259 | ** this controls ints. for both UART's | | | |
| 260 | ** we must take this into consideration. | | | |
| 261 | */ | | | |
| 262 | interrupt_reg = uart_mask; | | | |
| 263 | interrupt_reg.rx_rdy_b = DUART_ENABLE ; | | | |
| 264 | interrupt_reg.tx_rdy_b = DUART_ENABLE ; | | | |
| 265 | /* | | | |
| 266 | ** set interrupt reg, and save its mask this is a write only reg. | | | |
| 267 | */ | | | |
| 268 | uart_mask = interrupt_reg; | | | |
| 269 | *p_interrupt_reg = interrupt_reg; | | | |
| 270 | } | | | |
| 271 | /* UART BREAK RELATED FUNCTIONS */ | | | |
| 272 | uart_rx_brk() | | | |
| 273 | { | | | |
| 274 | int nvabaud; | | | |
| 275 | lm_get_nvaram_bauda(&nvabaud); | | | |
| 276 | if ((nvabaud & 0xf0) != 0x10) | | | |
| 277 | return; | | | |
| 278 | uart_baud(get_next_baud(lm_uartb_baud)); | | | |
| 279 | } | | | |
| 280 | uart_tx_brk() | | | |
| 281 | { | | | |
| 282 | int nvabaud; | | | |
| 283 | lm_get_nvaram_baudb(&nvabaud); | | | |
| 284 | if ((nvabaud & 0xf0) != 0x10) | | | |
| 285 | return; | | | |
| 286 | uart_baud(get_next_baud(lm_uartb_baud)); | | | |
| 287 | } | | | |
| 288 | static | | | |
| 289 | get_next_baud(baud) | | | |
| 290 | { | | | |
| 291 | switch (baud) { | | | |
| 292 | default: | | | |
| 293 | case BAUD_1200: | | | |
| 294 | return BAUD_2400; | | | |
| 295 | case BAUD_2400: | | | |
| 296 | return BAUD_9600; | | | |
| 297 | case BAUD_9600: | | | |
| 298 | return BAUD_1200; | | | |
| 299 | } | | | |
| 300 | } | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/eprom.c

DATE 5/23/89
TIME 4:42:11 pm

PAGE #
1/8

| LINE # | SOURCE TEXT |
|--------|---|
| 1 | /* SCCS_ID: eprom.c rev 3.1, 4/24/89 at 07:46:33 */ |
| 2 | /* |
| 3 | ** EEprom Access routines |
| 4 | */ |
| 5 | #include "common.h" |
| 6 | #include "mod_def.h" |
| 7 | #include "magic.h" |
| 8 | #include "pel.h" |
| 9 | #include "seeprom.h" |
| 10 | #include "ls_rd_wr.h" |
| 11 | #define REG_ADDRESS 8 |
| 12 | #define REG_DATA 0xa |
| 13 | |
| 14 | |
| 15 | extern u_char diag_dab; |
| 16 | u_char debug_eprom = 0; |
| 17 | static unsigned short reset[] = { |
| 18 | SET_CLK_0, |
| 19 | SET_CS_0, |
| 20 | SET_CS_1, |
| 21 | SET_DATA_IN_LOW, |
| 22 | TOG_CLK, |
| 23 | NOP, |
| 24 | NOP, |
| 25 | NOP, |
| 26 | NOP, |
| 27 | NOP, |
| 28 | NOP, |
| 29 | NOP, |
| 30 | NOP, |
| 31 | NOP, |
| 32 | NOP, |
| 33 | NOP, |
| 34 | NOP, |
| 35 | NOP, |
| 36 | NOP, |
| 37 | NOP, |
| 38 | NOP, |
| 39 | NOP, |
| 40 | NOP, |
| 41 | NOP, |
| 42 | NOP, |
| 43 | NOP, |
| 44 | NOP, |
| 45 | NOP, |
| 46 | NOP, |
| 47 | NOP, |
| 48 | NOP, |
| 49 | NO_TOG_CLK, |
| 50 | SET_CS_0, |
| 51 | SET_CLK_0, |
| 52 | DONE ; |
| 53 | static unsigned short read_command[] = { |
| 54 | SET_CLK_0, |
| 55 | SET_CS_1, |
| 56 | TOG_CLK, |
| 57 | LITERAL + 1, |
| 58 | START_OPCODE, |
| 59 | LITERAL + 2, |
| 60 | EEPROM_READ, |
| 61 | LITERAL + 6, |
| 62 | 0, |
| 63 | SET_DATA_IN_LOW, /* THIS IS WHERE THE ADDRESS GOES */ |
| 64 | NOP, |
| 65 | READ_DATA, |
| 66 | SET_CS_0, |
| 67 | SET_CS_0, |
| 68 | SET_CLK_0, |
| 69 | NO_TOG_CLK, |
| 70 | DONE ; |
| 71 | static unsigned short write_command[] = { |
| 72 | SET_CLK_0, |
| 73 | SET_CS_1, |
| 74 | TOG_CLK, |
| 75 | LITERAL + 1, |
| 76 | START_OPCODE, |
| 77 | LITERAL + 2, |
| 78 | EEPROM_WRITE, |
| 79 | LITERAL + 6, |
| 80 | 0, |
| 81 | LITERAL + 16, /* THIS IS WHERE THE ADDRESS GOES */ |
| 82 | 0, |
| 83 | LITERAL + 16, /* THIS IS WHERE THE DATA GOES */ |
| 84 | SET_CS_0, |
| 85 | SET_CS_0, |
| 86 | SET_CS_1, |
| 87 | /* WAIT OUT LOW, */ |
| 88 | WAIT_OUT_HIGH, |
| 89 | SET_CS_0, |
| 90 | NO_TOG_CLK, |
| 91 | DONE ; |
| 92 | static unsigned short erase_command[] = { |
| 93 | SET_CLK_0, |
| 94 | SET_CS_1, |
| 95 | TOG_CLK, |
| 96 | LITERAL + 1, |
| 97 | START_OPCODE, |
| 98 | LITERAL + 2, |
| 99 | EEPROM_ERASE, |
| 100 | LITERAL + 6, |
| 101 | 0, |
| 102 | SET_DATA_IN_LOW, /* THIS IS WHERE THE ADDRESS GOES */ |
| 103 | NOP, |
| 104 | NOP, |
| 105 | SET_CS_0, |
| 106 | SET_CS_0, |
| 107 | SET_CS_1, |
| 108 | /* WAIT OUT LOW, */ |
| 109 | WAIT_OUT_HIGH, |
| 110 | SET_CS_0, |
| 111 | NO_TOG_CLK, |
| 112 | DONE ; |
| 113 | |
| 114 | static unsigned short write_enable[] = { |
| 115 | SET_CLK_0, |
| 116 | SET_CS_1, |
| 117 | TOG_CLK, |
| 118 | LITERAL + 1, |
| 119 | START_OPCODE, |
| 120 | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/eprom.c | DATE 5/23/89 | PAGE # 2/9 |
|--|--|------------------------------|-----------------|---------------|
| LINE # | SOURCE TEXT | | | |
| 121 | LITERAL + 2, | | | |
| 122 | EEPROM_ALL, | | | |
| 123 | LITERAL + 6, | | | |
| 124 | EEPROM_END, | | | |
| 125 | SET_CS_0, | | | |
| 126 | NOP, | | | |
| 127 | NOP, | | | |
| 128 | NOP, | | | |
| 129 | NO_TOC_CLK, | | | |
| 130 | DONE ; | | | |
| 131 | | | | |
| 132 | static unsigned short write_disable[] = { | | | |
| 133 | SET_CLK_0, | | | |
| 134 | SET_CS_1, | | | |
| 135 | TOC_CLK, | | | |
| 136 | LITERAL + 1, | | | |
| 137 | START_OPCODE, | | | |
| 138 | LITERAL + 2, | | | |
| 139 | EEPROM_ALL, | | | |
| 140 | LITERAL + 6, | | | |
| 141 | EEPROM_ENDS, | | | |
| 142 | SET_CS_0, | | | |
| 143 | NOP, | | | |
| 144 | NOP, | | | |
| 145 | NOP, | | | |
| 146 | NO_TOC_CLK, | | | |
| 147 | DONE ; | | | |
| 148 | | | | |
| 149 | static unsigned short write_all_command[] = { | | | |
| 150 | SET_CLK_0, | | | |
| 151 | SET_CS_1, | | | |
| 152 | TOC_CLK, | | | |
| 153 | LITERAL + 1, | | | |
| 154 | START_OPCODE, | | | |
| 155 | LITERAL + 2, | | | |
| 156 | EEPROM_ALL, | | | |
| 157 | LITERAL + 6, | | | |
| 158 | EEPROM_ENDS, | | | |
| 159 | LITERAL + 16, | | | |
| 160 | 0 /* THIS IS WHERE THE DATA DOES */ | | | |
| 161 | SET_CS_0, | | | |
| 162 | SET_CS_0, | | | |
| 163 | SET_CS_1, | | | |
| 164 | /* WAIT OUT LOW, */ | | | |
| 165 | WAIT_OUT_HIGH, | | | |
| 166 | SET_CS_0, | | | |
| 167 | NO_TOC_CLK, | | | |
| 168 | DONE ; | | | |
| 169 | | | | |
| 170 | | | | |
| 171 | static unsigned short erase_all_command[] = { | | | |
| 172 | SET_CLK_0, | | | |
| 173 | SET_CS_1, | | | |
| 174 | TOC_CLK, | | | |
| 175 | LITERAL + 1, | | | |
| 176 | START_OPCODE, | | | |
| 177 | LITERAL + 2, | | | |
| 178 | EEPROM_ALL, | | | |
| 179 | LITERAL + 6, | | | |
| 180 | EEPROM_ENDS, | | | |
| 181 | NOP, | | | |
| 182 | NOP, | | | |
| 183 | SET_CS_0, | | | |
| 184 | SET_CS_0, | | | |
| 185 | SET_CS_1, | | | |
| 186 | /* WAIT OUT LOW, */ | | | |
| 187 | WAIT_OUT_HIGH, | | | |
| 188 | SET_CS_0, | | | |
| 189 | NO_TOC_CLK, | | | |
| 190 | DONE | | | |
| 191 | }; | | | |
| 192 | | | | |
| 193 | DAS EEPROM dab a2, | | | |
| 194 | extern u_short is_eprom_access(); | | | |
| 195 | u_short access_eprom(); | | | |
| 196 | u_short write_eprom(), read_eprom(); | | | |
| 197 | void disable_eprom(), enable_eprom(), erase_all_eprom(); | | | |
| 198 | | | | |
| 199 | static void | | | |
| 200 | memory_copy(a1, a2, count) | | | |
| 201 | register u_long *a1; | | | |
| 202 | register u_long *a2; | | | |
| 203 | register short count; | | | |
| 204 | { | | | |
| 205 | register short remain = count & 3; | | | |
| 206 | { | | | |
| 207 | /* | | | |
| 208 | ** divide by 4, | | | |
| 209 | ** how many long moves we need to perform | | | |
| 210 | */ | | | |
| 211 | count >>= 2; | | | |
| 212 | while(--count >= 0) { | | | |
| 213 | *a1++ = *a2++; | | | |
| 214 | } | | | |
| 215 | /* | | | |
| 216 | ** clean up, by moving remaining bytes | | | |
| 217 | */ | | | |
| 218 | while(--remain >= 0) { | | | |
| 219 | *(u_char *)a1++ = *(u_char *)a2++; | | | |
| 220 | } | | | |
| 221 | } | | | |
| 222 | | | | |
| 223 | #ifdef DEMOC | | | |
| 224 | static u_long error = 0; | | | |
| 225 | main() | | | |
| 226 | { | | | |
| 227 | u_short eedata; | | | |
| 228 | { | | | |
| 229 | u_char ee_number = 2; | | | |
| 230 | erase_command[REG_ADDRESS] = 0x22; | | | |
| 231 | write_command[REG_DATA] = 0x44; | | | |
| 232 | write_command[REG_ADDRESS] = 0x22; | | | |
| 233 | printf("erase"); | | | |
| 234 | (void) access_eprom(ee_number, &erase_command[0], &edata); | | | |
| 235 | printf("write"); | | | |
| 236 | (void) access_eprom(ee_number, &write_command[0], &edata); | | | |
| 237 | } | | | |
| 238 | test_eprom(ee_number) | | | |
| 239 | u_long ee_number; | | | |
| 240 | { | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/eeeprom.c

DATE 5/23/89
TIME 4:42:11 pm

PAGE #
3/10

SOURCE TEXT

```

241 printf("test_eeeprom\n");
242 if( !m_eeeprom_access( (char *)dab_e2, (u_long) 0, ee_number, (u_long) sizeof( dab_e2 ), MEMORY_VALIDATE, &error) == FAILURE)
243 {
244     printf("INVALID eeprom\n");
245     if( !m_eeeprom_access( (char *)dab_e2, (u_long) 0, ee_number, (u_long) sizeof( dab_e2 ), MEMORY_INIT, &error) == FAILURE)
246     {
247         printf("ERROR INITIALIZING EEPROM\n");
248     }
249 }
250 if( !m_eeeprom_access( (char *)dab_e2, (u_long) 0, ee_number, (u_long) sizeof( dab_e2 ), MEMORY_VALIDATE, &error) == FAILURE)
251 {
252     printf("ERROR RE-validating EEPROM\n");
253 }
254 if( !m_eeeprom_access( (char *)dab_e2, (u_long) 0, ee_number, (u_long) sizeof( dab_e2 ), MEMORY_READ, &error) == FAILURE)
255 {
256     printf("ERROR READING\n");
257 }
258 /*
259 if( !m_eeeprom_access( (char *)dab_e2, (u_long) 0, ee_number, (u_long) sizeof( dab_e2 ), MEMORY_WRITE, &error) == FAILURE)
260 {
261     printf("ERROR WRITING\n");
262 }
263 */
264 if( !m_eeeprom_access( (char *)dab_e2, (u_long) 0, ee_number, (u_long) sizeof( dab_e2 ), MEMORY_VALIDATE, &error) == FAILURE)
265 {
266     printf("ERRORSXXXXXXXXXXXX\n");
267 }
268 printf("SUCCESSFULLY TESTED EEPROM\n");
269 }
270 }
271 #endif
272 void
273 enable_eeeprom( ee_number )
274 u_char ee_number;
275 {
276     u_short eedata;
277     (void)access_eeeprom(ee_number,&write_enable[ 0 ], &edata);
278 }
279 void
280 erase_all_eeeprom( ee_number )
281 u_char ee_number;
282 {
283     u_short eedata;
284     enable_eeeprom( ee_number );
285     (void)access_eeeprom(ee_number,&erase_all_command[ 0 ], &edata);
286     disable_eeeprom( ee_number );
287 }
288 void
289 disable_eeeprom( ee_number )
290 u_char ee_number;
291 {
292     u_short eedata;
293     (void)access_eeeprom(ee_number,&write_disable[ 0 ], &edata);
294 }
295 u_short
296 write_eeeprom( ee_number, address, data )
297 u_char ee_number;
298 u_char address;
299 u_short data;
300 {
301     u_short v_cmd[ sizeof(write_command)], e_cmd[ sizeof(erase_command)];
302     u_short eedata;
303     u_short status;
304     (void)memory_copy( (u_long *)e_cmd, (u_long *)erase_command, (short) sizeof(erase_command));
305     (void)memory_copy( (u_long *)v_cmd, (u_long *)write_command, (short) sizeof(write_command));
306     e_cmd[ REG_ADDRESS ] = address;
307     v_cmd[ REG_ADDRESS ] = address;
308     v_cmd[ REG_DATA ] = data;
309     if( access_eeeprom(ee_number, &write_enable[ 0 ], &edata) == FAILURE) return( FAILURE);
310     if( access_eeeprom(ee_number, &e_cmd[ 0 ], &edata) == FAILURE) return( FAILURE);
311     if( access_eeeprom(ee_number, &v_cmd[ 0 ], &edata) == FAILURE) return( FAILURE);
312     if( access_eeeprom(ee_number, &write_disable[ 0 ], &edata) == FAILURE) return( FAILURE);
313     if( read_eeeprom( ee_number, address, &status ) != data)
314     {
315         if( debug_eeeprom == 1)
316             printf("error reading ee_number tx address tx expected data tx", ee_number, address, data);
317         return( FAILURE );
318     }
319     return( status );
320 }
321 u_short
322 read_eeeprom( ee_number, address, status )
323 u_short *status;
324 u_char ee_number;
325 u_char address;
326 {
327     u_short r_cmd[ sizeof(read_command)], eedata;
328     memory_copy( (u_long *)r_cmd, (u_long *)read_command,
329                 (short) sizeof(read_command));
330     r_cmd[ REG_ADDRESS ] = address;
331     *status = access_eeeprom(ee_number, &r_cmd[ 0 ], &edata);
332     return( eedata );
333 }
334 #define ACCESS_TIMEOUT 1000
335 u_short
336 access_eeeprom( ee_number, array, data )
337 u_char ee_number;
338 unsigned short array[];
339 register unsigned short *data;
340 {
341     register PEL *pel_access_reg;
342     register unsigned short *parray = &array[0];
343     register unsigned char skip = FALSE;
344     register unsigned char count = 16, toggle = 0;
345     register int timeout = 0;
346     *data = 0;
347     pel_access_reg = (PEL *)
348         (pel_addr(((ee_number & 0x1f) >> 1), (ee_number & 7)));
349     /*
350     ** if EEDOUT is stuck low, return FAILURE.
351     */

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/eprom.c | DATE 5/23/89 | PAGE # 4/11 |
|--|--|--|-----------------|----------------|
| LINE # | | SOURCE TEXT | | |
| 361 | | if(read_out(pel_access_reg) == 0) | | |
| 362 | | { | | |
| 363 | | printf("ERROR in stack low\n"); | | |
| 364 | | return(FAILURE); | | |
| 365 | | } | | |
| 366 | | while (*parray != DONE) | | |
| 367 | | { | | |
| 368 | | if (timeout >= ACCESS_TIMEOUT) break; | | |
| 369 | | { | | |
| 370 | | /* | | |
| 371 | | skip for diag dab | | |
| 372 | | otherwise, check active bit. | | |
| 373 | | */ | | |
| 374 | | if(diag_dab == 0 && pel_access_reg->car.bit.active == 0) | | |
| 375 | | { | | |
| 376 | | set_cs(pel_access_reg, 0); | | |
| 377 | | set_clk(pel_access_reg, 0); | | |
| 378 | | write_data(pel_access_reg, 0); | | |
| 379 | | return(FAILURE); | | |
| 380 | | } | | |
| 381 | | if(toggle) | | |
| 382 | | { | | |
| 383 | | if(skip == FALSE) | | |
| 384 | | set_clk(pel_access_reg, 0); | | |
| 385 | | skip = FALSE; | | |
| 386 | | } | | |
| 387 | | if(*parray & LITERAL) | | |
| 388 | | { | | |
| 389 | | if(*parray == LITERAL) | | |
| 390 | | { | | |
| 391 | | if(count != 0) | | |
| 392 | | { | | |
| 393 | | if(count > (*parray - LITERAL)) | | |
| 394 | | count = *parray - LITERAL; | | |
| 395 | | count--; | | |
| 396 | | write_data(pel_access_reg, | | |
| 397 | | (*parray + 1) >> count & 1); | | |
| 398 | | } | | |
| 399 | | else | | |
| 400 | | { | | |
| 401 | | count = 16; | | |
| 402 | | parray++; | | |
| 403 | | parray++; | | |
| 404 | | skip = TRUE; | | |
| 405 | | continue; | | |
| 406 | | } | | |
| 407 | | } | | |
| 408 | | } | | |
| 409 | | else | | |
| 410 | | { | | |
| 411 | | switch(*parray++) | | |
| 412 | | { | | |
| 413 | | case SET_CLK_0: | | |
| 414 | | set_clk(pel_access_reg, 0); | | |
| 415 | | break; | | |
| 416 | | case SET_CLK_1: | | |
| 417 | | set_clk(pel_access_reg, 1); | | |
| 418 | | break; | | |
| 419 | | case TOG_CLK: | | |
| 420 | | toggle = 1; | | |
| 421 | | break; | | |
| 422 | | case NO_TOG_CLK: | | |
| 423 | | toggle = 0; | | |
| 424 | | break; | | |
| 425 | | case SET_CS_0: | | |
| 426 | | set_cs(pel_access_reg, 0); | | |
| 427 | | break; | | |
| 428 | | case SET_CS_1: | | |
| 429 | | set_cs(pel_access_reg, 1); | | |
| 430 | | break; | | |
| 431 | | case READ_DATA: | | |
| 432 | | if(count--) | | |
| 433 | | { | | |
| 434 | | parray--; | | |
| 435 | | *data = (*data << 1) | | |
| 436 | | + read_out(pel_access_reg); | | |
| 437 | | } | | |
| 438 | | else | | |
| 439 | | count = 16; | | |
| 440 | | break; | | |
| 441 | | case WRITE_DATA: | | |
| 442 | | if(count--) | | |
| 443 | | { | | |
| 444 | | parray--; | | |
| 445 | | write_data(pel_access_reg, | | |
| 446 | | *data & (1 << (count - 1)); | | |
| 447 | | } | | |
| 448 | | else | | |
| 449 | | count = 16; | | |
| 450 | | break; | | |
| 451 | | case WAIT_OUT_LOW: | | |
| 452 | | if(read_out(pel_access_reg) == 1) { | | |
| 453 | | ++timeout; | | |
| 454 | | parray--; | | |
| 455 | | } | | |
| 456 | | break; | | |
| 457 | | case WAIT_OUT_HIGH: | | |
| 458 | | if(read_out(pel_access_reg) == 0) { | | |
| 459 | | ++timeout; | | |
| 460 | | parray--; | | |
| 461 | | } | | |
| 462 | | break; | | |
| 463 | | case WAIT_IN_LOW: | | |
| 464 | | if(read_in(pel_access_reg) == 1) { | | |
| 465 | | ++timeout; | | |
| 466 | | parray--; | | |
| 467 | | } | | |
| 468 | | break; | | |
| 469 | | case WAIT_IN_HIGH: | | |
| 470 | | if(read_in(pel_access_reg) == 0) { | | |
| 471 | | ++timeout; | | |
| 472 | | parray--; | | |
| 473 | | } | | |
| 474 | | break; | | |
| 475 | | case SET_DATA_IN_LOW: | | |
| 476 | | write_data(pel_access_reg, 0); | | |
| 477 | | skip = TRUE; | | |
| 478 | | continue; | | |
| 479 | | } | | |
| 480 | | case NOP: | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/eprom.c | DATE 5/23/89 | PAGE # 5/12 |
|--|--|--|-----------------|----------------|
| LINE # | | SOURCE TEXT | | |
| 481 | | break; | | |
| 482 | | } | | |
| 483 | | if(toggle) | | |
| 484 | | { | | |
| 485 | | set_clk(pel_access_reg, 1); | | |
| 486 | | } | | |
| 487 | | } | | |
| 488 | | return (timeout < ACCESS_TIMEOUT)? SUCCESS: FAILURE; | | |
| 489 | | } | | |
| 490 | | } | | |
| 491 | | | | |
| 492 | | set_clk(pel_access_reg, 1) | | |
| 493 | | PEL "pel_access_reg, | | |
| 494 | | u_char i; | | |
| 495 | | { | | |
| 496 | | #ifdef DEBUG | | |
| 497 | | printf("\nCLK=td", i); | | |
| 498 | | #else | | |
| 499 | | #endif HOST | | |
| 500 | | if(i) | | |
| 501 | | write_loc_short(pel_access_reg->car.bit, read_loc_short(pel_access_reg->car.bit) 0x40); | | |
| 502 | | else | | |
| 503 | | write_loc_short(pel_access_reg->car.bit, read_loc_short(pel_access_reg->car.bit) & 0xbf); | | |
| 504 | | #else | | |
| 505 | | pel_access_reg->car.bit.eeprom_clk = 1; | | |
| 506 | | #endif | | |
| 507 | | #endif | | |
| 508 | | } | | |
| 509 | | set_cs(pel_access_reg, 1) | | |
| 510 | | PEL "pel_access_reg, | | |
| 511 | | u_char i; | | |
| 512 | | { | | |
| 513 | | #ifdef DEBUG | | |
| 514 | | printf("\nCS=td", i); | | |
| 515 | | #else | | |
| 516 | | #endif HOST | | |
| 517 | | if(i) | | |
| 518 | | write_loc_short(pel_access_reg->car.bit, read_loc_short(pel_access_reg->car.bit) 0x80); | | |
| 519 | | else | | |
| 520 | | write_loc_short(pel_access_reg->car.bit, read_loc_short(pel_access_reg->car.bit) & 0x7f); | | |
| 521 | | #else | | |
| 522 | | pel_access_reg->car.bit.eeprom_sel = 1; | | |
| 523 | | #endif | | |
| 524 | | #endif | | |
| 525 | | } | | |
| 526 | | read_out(pel_access_reg) | | |
| 527 | | PEL "pel_access_reg, | | |
| 528 | | { | | |
| 529 | | #ifdef DEBUG | | |
| 530 | | printf("\nRead out", | | |
| 531 | | if(getchar() == '1') | | |
| 532 | | { | | |
| 533 | | return(1); | | |
| 534 | | } | | |
| 535 | | return 0; | | |
| 536 | | #else | | |
| 537 | | #endif HOST | | |
| 538 | | if(read_loc_short(pel_access_reg->car.bit) & 0x100) | | |
| 539 | | #else | | |
| 540 | | if(pel_access_reg->car.bit.eeprom_out == 1) | | |
| 541 | | #endif | | |
| 542 | | { | | |
| 543 | | return(1); | | |
| 544 | | } | | |
| 545 | | return 0; | | |
| 546 | | #endif | | |
| 547 | | } | | |
| 548 | | read_in(pel_access_reg) | | |
| 549 | | PEL "pel_access_reg, | | |
| 550 | | { | | |
| 551 | | #ifdef DEBUG | | |
| 552 | | printf("\nRead in", | | |
| 553 | | if(getchar() == '1') | | |
| 554 | | { | | |
| 555 | | return(1); | | |
| 556 | | } | | |
| 557 | | return 0; | | |
| 558 | | #else | | |
| 559 | | #endif HOST | | |
| 560 | | if(read_loc_short(pel_access_reg->car.bit) & 0x20) | | |
| 561 | | #else | | |
| 562 | | if(pel_access_reg->car.bit.eeprom_in == 1) | | |
| 563 | | #endif | | |
| 564 | | { | | |
| 565 | | return(1); | | |
| 566 | | } | | |
| 567 | | return 0; | | |
| 568 | | #endif | | |
| 569 | | } | | |
| 570 | | write_data(pel_access_reg, 1) | | |
| 571 | | PEL "pel_access_reg, | | |
| 572 | | unsigned char i; | | |
| 573 | | { | | |
| 574 | | #ifdef DEBUG | | |
| 575 | | printf("\nwrite data tx", i); | | |
| 576 | | #else | | |
| 577 | | #endif HOST | | |
| 578 | | if(i) | | |
| 579 | | write_loc_short(pel_access_reg->car.bit, read_loc_short(pel_access_reg->car.bit) 0x20); | | |
| 580 | | else | | |
| 581 | | write_loc_short(pel_access_reg->car.bit, read_loc_short(pel_access_reg->car.bit) & 0xdf); | | |
| 582 | | #else | | |
| 583 | | pel_access_reg->car.bit.eeprom_in = 1; | | |
| 584 | | #endif | | |
| 585 | | #endif | | |
| 586 | | } | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/fifo.c | DATE 5/23/89 | PAGE # 1/13 |
|--|---|-----------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCS ID: fifo.c rev 3.1, 4/24/89 at 07:46:42 */ | | | |
| 2 | /* | | | |
| 3 | /* fifo.c | | | |
| 4 | /* fifo handling for CPU board. | | | |
| 5 | /* | | | |
| 6 | /* | | | |
| 7 | #include "common.h" | | | |
| 8 | #include "fifo.h" | | | |
| 9 | #include "cpu.h" | | | |
| 10 | /* | | | |
| 11 | /* The event flag | | | |
| 12 | /* | | | |
| 13 | unsigned int event_flag; | | | |
| 14 | unsigned int fifo_group; | | | |
| 15 | /* | | | |
| 16 | /* fifos for the whole system | | | |
| 17 | /* | | | |
| 18 | struct fifo_type fifo[MAX_FIFOS + 1]; | | | |
| 19 | /* | | | |
| 20 | /* semaphores, and their IDs | | | |
| 21 | /* | | | |
| 22 | /* | | | |
| 23 | unsigned int fifo_0_id; | | | |
| 24 | unsigned int semaphore_id[MAX_FIFOS + 1]; | | | |
| 25 | unsigned int semaphore_count[MAX_FIFOS + 1]; | | | |
| 26 | /* | | | |
| 27 | /* headers for each fifo entry. | | | |
| 28 | /* | | | |
| 29 | struct fifo_header_type fifo_header[MAX_FIFO_ENTRIES]; | | | |
| 30 | /* | | | |
| 31 | /* | | | |
| 32 | /* Initialize fifo's and the headers. | | | |
| 33 | /* | | | |
| 34 | fifo_initialize() | | | |
| 35 | { | | | |
| 36 | register int i; | | | |
| 37 | unsigned int err; | | | |
| 38 | register struct fifo_header_type *pfifo_header = &fifo_header[0]; | | | |
| 39 | /* | | | |
| 40 | /* set the fifo data structure to zero. | | | |
| 41 | /* | | | |
| 42 | event_flag = 0; | | | |
| 43 | bzero(fifo, sizeof(fifo)); | | | |
| 44 | bzero(fifo_header, sizeof(fifo_header)); | | | |
| 45 | /* | | | |
| 46 | /* The headers organized in a queue, and associated with fifo 0 | | | |
| 47 | /* | | | |
| 48 | for(i = 0; i <= MAX_FIFO_ENTRIES-1; i++, pfifo_header++) | | | |
| 49 | pfifo_header->next = pfifo_header + 1; | | | |
| 50 | /* | | | |
| 51 | fifo[HEADER_FIFO].count = MAX_FIFO_ENTRIES; | | | |
| 52 | fifo[HEADER_FIFO].head = &fifo_header[0]; | | | |
| 53 | fifo[HEADER_FIFO].tail = &fifo_header[MAX_FIFO_ENTRIES - 1]; | | | |
| 54 | /* | | | |
| 55 | /* create semaphore 1 per fifo | | | |
| 56 | /* | | | |
| 57 | /* | | | |
| 58 | for(i = 0; i <= MAX_FIFOS; i++) | | | |
| 59 | { | | | |
| 60 | semaphore_id[i] = sc_create(0, 1, &err); | | | |
| 61 | semaphore_count[i] = 0; | | | |
| 62 | if(err) | | | |
| 63 | { | | | |
| 64 | sys_out("error creating SEMAPHORE\n"); | | | |
| 65 | return(FAILURE); | | | |
| 66 | } | | | |
| 67 | /* | | | |
| 68 | /* | | | |
| 69 | /* This avoids access to the array semaphore_id | | | |
| 70 | /* allocation of ID should be linear | | | |
| 71 | /* | | | |
| 72 | fifo_0_id = semaphore_id[0]; | | | |
| 73 | return(SUCCESS); | | | |
| 74 | } | | | |
| 75 | /* | | | |
| 76 | /* place a data buffer in a fifo(pfifo_entry->fifo_number) | | | |
| 77 | /* | | | |
| 78 | fifo_put(pfifo_entry); | | | |
| 79 | { | | | |
| 80 | register struct fifo_entry *pfifo_entry; | | | |
| 81 | /* | | | |
| 82 | register struct fifo_header_type *pfifo_header; | | | |
| 83 | register struct fifo_type *pfifo; | | | |
| 84 | register char fifo_no = pfifo_entry->fifo_no; | | | |
| 85 | register unsigned int *fifo_count = &semaphore_count[fifo_no]; | | | |
| 86 | int err = 0; | | | |
| 87 | /* | | | |
| 88 | /* do a sanity check | | | |
| 89 | /* | | | |
| 90 | if(fifo_no > MAX_FIFOS) return(FAILURE); | | | |
| 91 | /* | | | |
| 92 | /* grab a header | | | |
| 93 | /* | | | |
| 94 | /* | | | |
| 95 | /* speed up access to fifo | | | |
| 96 | /* | | | |
| 97 | pfifo = &fifo[HEADER_FIFO]; | | | |
| 98 | /* | | | |
| 99 | /* disable interrupts | | | |
| 100 | /* | | | |
| 101 | CPU_DISABLE_INTERRUPTS; | | | |
| 102 | /* | | | |
| 103 | /* are there any header buffers, for the ISR | | | |
| 104 | /* | | | |
| 105 | if(pfifo_entry->task == ISR_TASK) | | | |
| 106 | { | | | |
| 107 | if(!pfifo->count) | | | |
| 108 | { | | | |
| 109 | CPU_ENABLE_INTERRUPTS; | | | |
| 110 | return(FAILURE); | | | |
| 111 | } | | | |
| 112 | } | | | |
| 113 | else | | | |
| 114 | { | | | |
| 115 | /* | | | |
| 116 | /* are there any header buffers, for the TASK | | | |
| 117 | /* | | | |
| 118 | while(pfifo->count < MIN_TASK_ENTRIES) | | | |
| 119 | { | | | |
| 120 | event_flag = ((unsigned int) 1 << HEADER_FIFO); | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/fifo.c | DATE 5/23/89 | PAGE # 2/14 |
|--|---|-----------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 121 | CPU_DISABLE_INTERRUPTS; | | | |
| 122 | sc_spend(semaphore_id(HEADER_FIFO), 0, &err); | | | |
| 123 | if(err) return(FAILURE); | | | |
| 124 | CPU_DISABLE_INTERRUPTS; | | | |
| 125 | } | | | |
| 126 | /* | | | |
| 127 | */ | | | |
| 128 | /* get pointer to first entry | | | |
| 129 | */ | | | |
| 130 | pfifo_header = pfifo->head; | | | |
| 131 | /* | | | |
| 132 | /* having retrieved, update pointers and count | | | |
| 133 | */ | | | |
| 134 | if(--pfifo->count) | | | |
| 135 | pfifo->head = pfifo_header->next; | | | |
| 136 | else | | | |
| 137 | pfifo->head = pfifo->tail = 0; | | | |
| 138 | /* | | | |
| 139 | /* point to appropriate fifo | | | |
| 140 | */ | | | |
| 141 | pfifo = &fifo(fifo_no); | | | |
| 142 | /* | | | |
| 143 | /* set up the header | | | |
| 144 | */ | | | |
| 145 | pfifo_header->user = pfifo_entry->user; | | | |
| 146 | pfifo_header->next = 0; | | | |
| 147 | pfifo_header->data = pfifo_entry->data; | | | |
| 148 | pfifo_header->task = pfifo_entry->task; | | | |
| 149 | /* | | | |
| 150 | /* add to the appropriate queue | | | |
| 151 | */ | | | |
| 152 | if(pfifo->count) | | | |
| 153 | pfifo->tail->next = pfifo_header; | | | |
| 154 | else | | | |
| 155 | pfifo->head = pfifo_header; | | | |
| 156 | pfifo->tail = pfifo_header; | | | |
| 157 | pfifo->count++; | | | |
| 158 | /* | | | |
| 159 | /* if (*fifo_count) | | | |
| 160 | { | | | |
| 161 | /* semaphore_count(fifo_no)--; | | | |
| 162 | /* (*fifo_count)--; | | | |
| 163 | /* system_call = 1; | | | |
| 164 | CPU_DISABLE_INTERRUPTS; | | | |
| 165 | /* | | | |
| 166 | /* should we try and wakeup someone | | | |
| 167 | /* to avoid an array access -> sc_spend(semaphore_id(fifo_no), &err); | | | |
| 168 | /* use fifo_0_id. | | | |
| 169 | /* | | | |
| 170 | sc_spend(fifo_0_id + fifo_no, &err); | | | |
| 171 | /* | | | |
| 172 | if(err) | | | |
| 173 | return(FAILURE); | | | |
| 174 | CPU_DISABLE_INTERRUPTS; | | | |
| 175 | return(SUCCESS); | | | |
| 176 | } | | | |
| 177 | /* | | | |
| 178 | /* get a fifo entry | | | |
| 179 | */ | | | |
| 180 | fifo_get(pfifo_entry) | | | |
| 181 | struct fifo_entry *pfifo_entry; | | | |
| 182 | { | | | |
| 183 | register struct fifo_header_type *pfifo; | | | |
| 184 | register struct fifo_type *pfifo; | | | |
| 185 | register char fifo_no = pfifo_entry->fifo_no; | | | |
| 186 | int err = 0; | | | |
| 187 | int event = 0; | | | |
| 188 | /* | | | |
| 189 | /* if (fifo_no > MAX_FIFOS) return(FAILURE); | | | |
| 190 | /* | | | |
| 191 | /* set up pointer to appropriate fifo | | | |
| 192 | */ | | | |
| 193 | pfifo = &fifo(fifo_no); | | | |
| 194 | /* | | | |
| 195 | /* disable interrupts | | | |
| 196 | CPU_DISABLE_INTERRUPTS; | | | |
| 197 | /* | | | |
| 198 | /* while no entries stick around | | | |
| 199 | /* | | | |
| 200 | while(!pfifo->count) | | | |
| 201 | { | | | |
| 202 | semaphore_count(fifo_no)++; | | | |
| 203 | CPU_DISABLE_INTERRUPTS; | | | |
| 204 | /* | | | |
| 205 | /* To avoid an array access, use fifo_0_id | | | |
| 206 | /* | | | |
| 207 | sc_spend(fifo_0_id + fifo_no, 0, &err); | | | |
| 208 | if(err) return(FAILURE); | | | |
| 209 | CPU_DISABLE_INTERRUPTS; | | | |
| 210 | /* | | | |
| 211 | /* get an entry | | | |
| 212 | */ | | | |
| 213 | pfifo->head = pfifo->head; | | | |
| 214 | /* | | | |
| 215 | /* adjust pointers | | | |
| 216 | */ | | | |
| 217 | pfifo->head = pfifo->head->next; | | | |
| 218 | /* | | | |
| 219 | /* if no entry? tail = head = 0 | | | |
| 220 | /* | | | |
| 221 | if(! --pfifo->count) | | | |
| 222 | pfifo->tail = 0; | | | |
| 223 | /* | | | |
| 224 | /* prepare return data struct | | | |
| 225 | /* | | | |
| 226 | pfifo_entry->data = pfifo->data; | | | |
| 227 | pfifo_entry->user = pfifo->user; | | | |
| 228 | pfifo_entry->task = pfifo->task; | | | |
| 229 | /* | | | |
| 230 | /* return header to free queue | | | |
| 231 | /* | | | |
| 232 | pfifo = &fifo(HEADER_FIFO); | | | |
| 233 | /* | | | |
| 234 | /* adjust pointers and check | | | |
| 235 | /* | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/fifo.c | DATE 5/23/89 TIME 4:42:11 pm | PAGE # 3/15 |
|--|---|-----------------------------|---------------------------------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 241 | /* if any one waiting for this | | | |
| 242 | */ | | | |
| 243 | if(pfifo->count) | | | |
| 244 | fifo->tail->next = pfifo; | | | |
| 245 | else | | | |
| 246 | pfifo->head = pfifo; | | | |
| 247 | /* | | | |
| 248 | /* the tail points to the new entry | | | |
| 249 | */ | | | |
| 250 | pfifo->tail = pfifo; | | | |
| 251 | /* | | | |
| 252 | /* update count | | | |
| 253 | */ | | | |
| 254 | pfifo->count++; | | | |
| 255 | /* | | | |
| 256 | /* is any one waiting for a header | | | |
| 257 | */ | | | |
| 258 | if(event_flag & (1 << HEADER_FIFO)) | | | |
| 259 | event = 1; | | | |
| 260 | /* | | | |
| 261 | /* enable interrupts | | | |
| 262 | */ | | | |
| 263 | CPU_ENABLE_INTERRUPTS; | | | |
| 264 | /* | | | |
| 265 | /* post event to a flag | | | |
| 266 | */ | | | |
| 267 | if(event) | | | |
| 268 | { | | | |
| 269 | sc_spost(semaphore_id(HEADER_FIFO), terr); | | | |
| 270 | } | | | |
| 271 | /* | | | |
| 272 | /* done | | | |
| 273 | */ | | | |
| 274 | if(err) | | | |
| 275 | return(FAILURE); | | | |
| 276 | return(SUCCESS); | | | |
| 277 | } | | | |
| 278 | /* | | | |
| 279 | /* return number of elements in a fifo | | | |
| 280 | */ | | | |
| 281 | fifo_inquiry(pfifo_entry) | | | |
| 282 | struct fifo_entry *pfifo_entry; | | | |
| 283 | { | | | |
| 284 | register struct fifo_type *pfifo; | | | |
| 285 | register char fifo_no = pfifo_entry->fifo_no; | | | |
| 286 | if(fifo_no > MAX_FIFOS) return(0); | | | |
| 287 | /* | | | |
| 288 | /* set up pointer to appropriate fifo | | | |
| 289 | */ | | | |
| 290 | pfifo = &fifo(fifo_no); | | | |
| 291 | return pfifo->count; | | | |
| 292 | } | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/id.c | DATE 5/23/89 | PAGE # 1/16 |
|--|--|---|-----------------|----------------|
| LINE # | | SOURCE TEXT | | |
| 1 | | /* SCCS_ID: id.c rev 3.1, 4/24/89 at 07:46:44 */ | | |
| 2 | | #include "common.h" | | |
| 3 | | #include "ls_diags.h" | | |
| 4 | | #include "ls_afi.h" | | |
| 5 | | #include "id.h" | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | /* | | |
| 9 | | * Compute and verify checksum on ID FROM | | |
| 10 | | * Pass pointer to first byte in ID FROM. | | |
| 11 | | * Subsequent ID_NUM_BYTES-1 bytes are 4 bytes apart. | | |
| 12 | | * Algorithm: | | |
| 13 | | * initialize checksum to an arbitrary (but well-known) value | | |
| 14 | | * for each byte (including checksum) | | |
| 15 | | * circular left shift left checksum | | |
| 16 | | * add data byte | | |
| 17 | | * mask checksum to 3 bits | | |
| 18 | | * Returns computed checksum. | | |
| 19 | | */ | | |
| 20 | | int | | |
| 21 | | id_check(address) | | |
| 22 | | u_char *address, | | |
| 23 | | { | | |
| 24 | | register int checksum; | | |
| 25 | | register u_long byte_count; | | |
| 26 | | | | |
| 27 | | checksum= ID_CHECKSUM_INIT; | | |
| 28 | | | | |
| 29 | | for (byte_count= 0; byte_count < ID_NUM_BYTES; byte_count++) | | |
| 30 | | { | | |
| 31 | | checksum= (checksum << 1) + ((checksum & 0x80) >> 7); | | |
| 32 | | checksum+= *(address + 4 * byte_count); | | |
| 33 | | checksum+= 0xFF; | | |
| 34 | | } | | |
| 35 | | return(checksum); | | |
| 36 | | } | | |
| 37 | | | | |
| 38 | | /* | | |
| 39 | | * Load as ID FROM into a character buffer | | |
| 40 | | * address is the physical byte address of the first byte in the ID FROM. | | |
| 41 | | * buffer is the byte address of the first byte in a ID_NUM_BYTES buffer. | | |
| 42 | | * In practice, buffer is really an appropriate ID_FROM_XXX structure, and | | |
| 43 | | * a pointer to it is passed, cast to (u_char *). | | |
| 44 | | */ | | |
| 45 | | void | | |
| 46 | | id_load(address, buffer) | | |
| 47 | | u_char *address, | | |
| 48 | | u_char *buffer, | | |
| 49 | | { | | |
| 50 | | register int byte_count; | | |
| 51 | | for (byte_count= 0; byte_count < ID_NUM_BYTES; byte_count++) | | |
| 52 | | { | | |
| 53 | | *buffer= *address; | | |
| 54 | | buffer++; | | |
| 55 | | address+= 4; | | |
| 56 | | } | | |
| 57 | | } | | |
| 58 | | | | |
| 59 | | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lance.c | DATE 5/23/89 | PAGE # 1/17 |
|--|--|---|-----------------|----------------|
| LINE # | | SOURCE TEXT | | |
| 1 | | /* SCCS ID: lance.c rev 3.2, 4/23/89 at 16:44:16 */ | | |
| 2 | | #include "common.h" | | |
| 3 | | #include "cpu.h" | | |
| 4 | | #include "vrx.h" | | |
| 5 | | #include "lance.h" | | |
| 6 | | #include "id.h" | | |
| 7 | | #include "network.h" | | |
| 8 | | #include "mod_err.h" | | |
| 9 | | #include "svekm.h" | | |
| 10 | | #include "lm_xd_vr.h" | | |
| 11 | | #include "arp.h" | | |
| 12 | | #include "arp.h" | | |
| 13 | | #define SECOND 200 | | |
| 14 | | #define TIMEOUT 33 | | |
| 15 | | #define MAX_LANCE_ENABLE_POLLS 30 | | |
| 16 | | #define MAX_INITIALIZED_DONE_POLLS 5 | | |
| 17 | | /* | | |
| 18 | | * Memory for the LANCE initialization block. | | |
| 19 | | * Init_block_mem() must be word aligned. | | |
| 20 | | */ | | |
| 21 | | char init_block_mem(alignof(INIT_BLOCK)), | | |
| 22 | | /* | | |
| 23 | | u_long system_call; | | |
| 24 | | u_long rcv_lance_pkt_semaphore; | | |
| 25 | | static u_long alive_inet; | | |
| 26 | | /* | | |
| 27 | | u_char accept_arp_reply = 0; | | |
| 28 | | /* | | |
| 29 | | u_long modeler_inet; | | |
| 30 | | u_long modeler_inet_address; | | |
| 31 | | /* | | |
| 32 | | #ifdef BROKEN_HARDWARE | | |
| 33 | | char reinitialize_lance = 0; | | |
| 34 | | #endif | | |
| 35 | | #ifdef BROKEN_HARDWARE | | |
| 36 | | extern BOOT_STRUCT boot; | | |
| 37 | | /* | | |
| 38 | | * Required variable to allow the Lance diagnostics to use sendto() | | |
| 39 | | * with small (runt) packets. | | |
| 40 | | */ | | |
| 41 | | u_short allow_runt_packets = FALSE; | | |
| 42 | | /* | | |
| 43 | | * Structure definition and allocation of permanent memory | | |
| 44 | | * for the desired number of receive buffers. | | |
| 45 | | */ | | |
| 46 | | #define MAX_RECEIVE_BUFFERS 10 | | |
| 47 | | #define MAX_TRANSMIT_BUFFERS 10 | | |
| 48 | | typedef struct { | | |
| 49 | | char buf[MAX_RECEIVE_BUFFER_SIZE]; | | |
| 50 | | } RECV_BUF; | | |
| 51 | | /* | | |
| 52 | | static RECV_BUF lm_receive_buffers(MAX_RECEIVE_BUFFERS + MAX_USERS); | | |
| 53 | | /* | | |
| 54 | | static RECV_BUF *p_lm_rcv_buffers_extra(MAX_USERS); | | |
| 55 | | static RECV_MDE extra_rcv_desc_buffers(MAX_USERS); | | |
| 56 | | /* | | |
| 57 | | static unsigned char extra_rcv_desc_count = 0; | | |
| 58 | | static unsigned char in_extra_rcv_desc_ptr = 0; | | |
| 59 | | static unsigned char out_extra_rcv_desc_ptr = 0; | | |
| 60 | | /* | | |
| 61 | | static char too_many_buffers = 0; | | |
| 62 | | static char available_extra_buffer = (MAX_USERS - 1); | | |
| 63 | | /* | | |
| 64 | | * Note that the following character array space is 8 bytes larger than | | |
| 65 | | * what is really required in order to get space that is aligned on a | | |
| 66 | | * quadword (8 byte) boundary. The stupid SUN assembler has no means for | | |
| 67 | | * allowing me to specify this directly. | | |
| 68 | | */ | | |
| 69 | | /* | | |
| 70 | | * Also note that each entry mapped into the table must start on a | | |
| 71 | | * quadword (8 byte) boundary. This automatically happens because of | | |
| 72 | | * the sizeof("MDE") being equal to 8 bytes. | | |
| 73 | | */ | | |
| 74 | | static char rcv_desc_ring_memory(8 + MAX_RECEIVE_BUFFERS * sizeof(RECV_MDE)); | | |
| 75 | | static char trans_desc_ring_memory(8 + MAX_TRANSMIT_BUFFERS * sizeof(TRANS_MDE)); | | |
| 76 | | /* | | |
| 77 | | * Macro for returning a pointer value that is aligned on a quad-word boundary. | | |
| 78 | | */ | | |
| 79 | | #define QUAD_WORD_ALIGN(X) ((8 * (((u_long) (X) + 7) / 8))) | | |
| 80 | | /* | | |
| 81 | | * Pointers to keep track of our notion of the beginning, current and end | | |
| 82 | | * of the receive and transmit descriptor message rings. These pointers | | |
| 83 | | * are given values when the receive and transmit rings are initialized. | | |
| 84 | | */ | | |
| 85 | | static RECV_MDE *minimum_receive_pointer; | | |
| 86 | | static RECV_MDE *current_receive_pointer; | | |
| 87 | | static RECV_MDE *maximum_receive_pointer; | | |
| 88 | | static TRANS_MDE *minimum_transmit_pointer; | | |
| 89 | | static TRANS_MDE *current_transmit_pointer; | | |
| 90 | | static TRANS_MDE *maximum_transmit_pointer; | | |
| 91 | | /* | | |
| 92 | | static TRANS_MDE *lance_send_packet; | | |
| 93 | | /* | | |
| 94 | | extern u_short initialize_lance(); | | |
| 95 | | extern u_short set_up_control_and_status_registers(); | | |
| 96 | | /* | | |
| 97 | | extern void set_up_extra_buffers(); | | |
| 98 | | extern void memory_align_ring_descriptors(); | | |
| 99 | | extern void initialize_transmit_buffers(); | | |
| 100 | | extern void initialize_receive_buffers(); | | |
| 101 | | extern void output_routine(); | | |
| 102 | | extern void set_up_initialization_block(); | | |
| 103 | | static void memory_copy(); | | |
| 104 | | extern void lance_isr_receive(); | | |
| 105 | | /* | | |
| 106 | | extern u_short lance_receive(); | | |
| 107 | | extern u_short lance_transmit(); | | |
| 108 | | extern u_short process_transmit_interrupt(); | | |
| 109 | | extern u_short process_receive_interrupt(); | | |
| 110 | | /* | | |
| 111 | | int lm_intr_requested; /* global interrupt advisory flag */ | | |
| 112 | | /* | | |
| 113 | | * Properly reset the LANCE using the CPU control register bit. | | |
| 114 | | * Leaves it reset, or may be followed by turning off the lance reset | | |
| 115 | | * to provide a reset_lance() function. | | |
| 116 | | */ | | |
| 117 | | /* | | |
| 118 | | * Properly reset the LANCE using the CPU control register bit. | | |
| 119 | | * Leaves it reset, or may be followed by turning off the lance reset | | |
| 120 | | * to provide a reset_lance() function. | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lance.c | DATE 5/23/89 | PAGE # 2/18 |
|--|---|------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 121 | void | | | |
| 122 | shutdown_lance() | | | |
| 123 | { | | | |
| 124 | register cpu_control_reg_struct *control_reg; | | | |
| 125 | register u_long delay_count; | | | |
| 126 | } | | | |
| 127 | /* | | | |
| 128 | The LANCE may very well be doing something. In that event, we | | | |
| 129 | don't want to risk confusing its requester and possibly the CPU | | | |
| 130 | arbitrar, so we let it complete. In order to guarantee that it | | | |
| 131 | doesn't start any master cycles, turn off the lance enable bit. | | | |
| 132 | Then delay a generous amount of time to allow for a worst-case | | | |
| 133 | burst. These seminally require 6-bus, so we'll allow 50us. After | | | |
| 134 | all, we're not in THAT much of a hurry here. Then pound on the | | | |
| 135 | reset bit and hold it. Though the Am7990 specs just 2 clock cycles | | | |
| 136 | minimum low time (i.e. 200ns), we drive it for a more comfortable | | | |
| 137 | 5us. No real good reason. | | | |
| 138 | */ | | | |
| 139 | control_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG; | | | |
| 140 | control_reg->lance_enable = 0; | | | |
| 141 | delay_microseconds(delay_count, 50); | | | |
| 142 | control_reg->not_lance_reset = 0; | | | |
| 143 | delay_microseconds(delay_count, 5); | | | |
| 144 | } | | | |
| 145 | /* | | | |
| 146 | That packets are used only in diagnosis for loopback tests. | | | |
| 147 | We shut off this flag here just in case it got set spuriously. | | | |
| 148 | */ | | | |
| 149 | allow_rmt_packets = FALSE; | | | |
| 150 | } | | | |
| 151 | /* | | | |
| 152 | Shut down the LANCE, then allow it to come out of reset. | | | |
| 153 | */ | | | |
| 154 | void | | | |
| 155 | shutdown_and_reset_lance() | | | |
| 156 | { | | | |
| 157 | register cpu_control_reg_struct *control_reg; | | | |
| 158 | control_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG; | | | |
| 159 | shutdown_lance(); | | | |
| 160 | control_reg->not_lance_reset = 1; | | | |
| 161 | } | | | |
| 162 | u_short | | | |
| 163 | set_lance_ready_to_go() | | | |
| 164 | { | | | |
| 165 | RCV_RING *rcv_ring; | | | |
| 166 | TRANS_RING *trans_ring; | | | |
| 167 | in_intr_requested = 0; | | | |
| 168 | shutdown_and_reset_lance(); | | | |
| 169 | set_up_extra_buffers(); | | | |
| 170 | memory_align_ring_descriptors(&rcv_ring, &trans_ring); | | | |
| 171 | initialize_transmit_buffers(trans_ring); | | | |
| 172 | initialize_receive_buffers(rcv_ring); | | | |
| 173 | if (initialize_lance(rcv_ring, trans_ring) == FAILURE) | | | |
| 174 | return(FAILURE); | | | |
| 175 | return(SUCCESS); | | | |
| 176 | } | | | |
| 177 | void | | | |
| 178 | set_up_extra_buffers() | | | |
| 179 | { | | | |
| 180 | register i; | | | |
| 181 | for(i=0; i < MAX_USERS; i++) { | | | |
| 182 | p_lm_rcv_buffers_extra[i] = &lm_receive_buffers[i + MAX_RECEIVE_BUFFERS]; | | | |
| 183 | } | | | |
| 184 | } | | | |
| 185 | u_short | | | |
| 186 | start_lance() | | | |
| 187 | { | | | |
| 188 | u_short value; | | | |
| 189 | register u_short i; | | | |
| 190 | } | | | |
| 191 | /* | | | |
| 192 | Tell the LANCE to initialize itself. | | | |
| 193 | */ | | | |
| 194 | if (write_lance_car(SELECT_CSRO, INITIALIZE_START) == FAILURE) | | | |
| 195 | return(FAILURE); | | | |
| 196 | /* | | | |
| 197 | We must wait for the LANCE to finish its | | | |
| 198 | initialization before we continue processing. | | | |
| 199 | */ | | | |
| 200 | for (i=0; i<MAX_INITIALIZED_DONE_POLLS; ++i) { | | | |
| 201 | lm_delay(3); | | | |
| 202 | if (read_lance_car(SELECT_CSRO, &value) == FAILURE) | | | |
| 203 | return(FAILURE); | | | |
| 204 | if ((value & INITIALIZE_DONE) == INITIALIZE_DONE) | | | |
| 205 | break; | | | |
| 206 | } | | | |
| 207 | /* | | | |
| 208 | If the LANCE still hasn't initialized itself, give up. | | | |
| 209 | */ | | | |
| 210 | if ((value & INITIALIZE_DONE) != INITIALIZE_DONE) | | | |
| 211 | return(FAILURE); | | | |
| 212 | /* | | | |
| 213 | Let's get the show on the road. | | | |
| 214 | */ | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lance.c | DATE 5/23/89 TIME 4:42:12 pm | PAGE # 3/19 |
|--|---|------------------------------|---------------------------------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 241 | if (write_lance_csr(SELECT_CSR0, START_ACTIVITY) == FAILURE) | | | |
| 242 | return(Failure); | | | |
| 243 | /* | | | |
| 244 | * Things appear to work better (at all?) if we wait a little bit here. | | | |
| 245 | */ | | | |
| 246 | lm_delay(10); | | | |
| 247 | /* | | | |
| 248 | * Enable LANCE interrupts. | | | |
| 249 | */ | | | |
| 250 | if (write_lance_csr(SELECT_CSR0, INTERRUPT_ENABLE) == FAILURE) | | | |
| 251 | return(Failure); | | | |
| 252 | /* | | | |
| 253 | * Things appear to work better (at all?) if we wait a little bit here. | | | |
| 254 | */ | | | |
| 255 | lm_delay(10); | | | |
| 256 | return(SUCCESS); | | | |
| 257 | } | | | |
| 258 | /* | | | |
| 259 | * The following function is used by the CPU diagnostics to have the | | | |
| 260 | * LANCE read the initialization block but not start activity. It must | | | |
| 261 | * not use any VMT calls, e.g. lm_delay(). | | | |
| 262 | */ | | | |
| 263 | u_short | | | |
| 264 | start_lance_initialize_only() | | | |
| 265 | { | | | |
| 266 | u_short i; | | | |
| 267 | u_short value; | | | |
| 268 | register u_long bogus; | | | |
| 269 | /* | | | |
| 270 | * Tell the LANCE to initialize itself. | | | |
| 271 | */ | | | |
| 272 | if (write_lance_csr(SELECT_CSR0, (u_short)INITIALIZE_START) == FAILURE) | | | |
| 273 | return (FAILURE); | | | |
| 274 | /* | | | |
| 275 | * We must wait for the LANCE to finish its initialization before we | | | |
| 276 | * continue processing. | | | |
| 277 | */ | | | |
| 278 | for (i = 0; i < MAX_INITIALIZE_DONE_POLLS; ++i) | | | |
| 279 | { | | | |
| 280 | delay_milliseconds(bogus, 3); | | | |
| 281 | if (read_lance_csr(SELECT_CSR0, ivalue) == FAILURE) | | | |
| 282 | return (FAILURE); | | | |
| 283 | if ((value & INITIALIZE_DONE) == INITIALIZE_DONE) | | | |
| 284 | break; | | | |
| 285 | } | | | |
| 286 | /* | | | |
| 287 | * If the LANCE still hasn't initialized itself, give up. | | | |
| 288 | */ | | | |
| 289 | if ((value & INITIALIZE_DONE) != INITIALIZE_DONE) | | | |
| 290 | return (FAILURE); | | | |
| 291 | return(SUCCESS); | | | |
| 292 | } | | | |
| 293 | /* | | | |
| 294 | * Quad-word align the transmit and receive message descriptor rings. | | | |
| 295 | * This is a LANCE requirement that the SUN assembler can not provide. | | | |
| 296 | */ | | | |
| 297 | if ((u_long) trans_desc_ring_memory & 8 == 0) | | | |
| 298 | trans = (TRANS_NDE *) trans_desc_ring_memory; | | | |
| 299 | else trans = (TRANS_NDE *) QWORD_ALIGN(trans_desc_ring_memory); | | | |
| 300 | if ((u_long) recv_desc_ring_memory & 8 == 0) | | | |
| 301 | recv = (RCV_NDE *) recv_desc_ring_memory; | | | |
| 302 | else recv = (RCV_NDE *) QWORD_ALIGN(recv_desc_ring_memory); | | | |
| 303 | /* | | | |
| 304 | * Initialize transmit buffers (trans) | | | |
| 305 | */ | | | |
| 306 | void | | | |
| 307 | initialize_transmit_buffers(trans) | | | |
| 308 | { | | | |
| 309 | register TRANS_NDE *trans; | | | |
| 310 | register | | | |
| 311 | u_short i; | | | |
| 312 | /* | | | |
| 313 | * Set an unchanging pointer to the beginning of the transmit | | | |
| 314 | * message descriptor ring so that we know where to wrap around | | | |
| 315 | * to as more and more messages are sent (see lance_transmit()). | | | |
| 316 | */ | | | |
| 317 | minimize_transmit_pointer = trans; | | | |
| 318 | /* | | | |
| 319 | * Set curr. as opposed to the LANCE's notion of what the current | | | |
| 320 | * transmit message descriptor is. | | | |
| 321 | */ | | | |
| 322 | current_transmit_pointer = trans; | | | |
| 323 | lance_send_packet = trans; | | | |
| 324 | /* | | | |
| 325 | * for(i=0; i<MAX_TRANSMIT_BUFFERS; ++i) { | | | |
| 326 | /* No other fields in the transmit message descriptor */ | | | |
| 327 | /* need be initialized at this time. They are all set */ | | | |
| 328 | /* when the to be transmitted packet buffer is set up. */ | | | |
| 329 | { | | | |
| 330 | trans->must_be_ones = 0xf; | | | |
| 331 | trans->own_buffer = HOST_BUFFER_OWNERSHIP; | | | |
| 332 | ++trans; /* advance to the next transmit descriptor. */ | | | |
| 333 | } | | | |
| 334 | /* | | | |
| 335 | * Set an unchanging pointer to the end of the transmit | | | |
| 336 | * message descriptor ring so that we know when to wrap around | | | |
| 337 | * as more and more messages are sent (see lance_transmit()). | | | |
| 338 | */ | | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

os/lance.c

#

DATE 5/23/89

PAGE #

TIME 4:42:12 pm

4/20

```

LINE # SOURCE TEXT
361     maximum_transmit_pointer = trans;
362 }
363
364
365 void
366 initialize_receive_buffers(recv)
367 register RECV_MODE *recv;
368 {
369     register u_short i;
370
371     /*
372     * Set an unchanging pointer to the beginning of the receive
373     * message descriptor ring so that we know where to wrap around
374     * to as more and more messages are received (see lance_receive()).
375     */
376     minimum_receive_pointer = recv;
377
378     /*
379     * Set our, as opposed to the LANCE's, notion of what the current
380     * receive message descriptor is.
381     */
382     current_receive_pointer = recv;
383
384     for(i=0; i<MAX_RECEIVE_BUFFERS; ++i) {
385         /* Set receive message descriptor 0 */
386         recv->low_buffer_address = (u_short) &lance_receive_buffers[i];
387
388         /* Set receive message descriptor 1 */
389         recv->own_buffer = LANCE_BUFFER_OWNERSHIP;
390         recv->error_summary = 0;
391         recv->framing_error = 0;
392         recv->overflow_error = 0;
393         recv->crc_error = 0;
394         recv->buffer_error = 0;
395         recv->start_of_packet = 0;
396         recv->end_of_packet = 0;
397         recv->high_buffer_address = (unsigned) &lance_receive_buffers[i] >> 16;
398
399         /* Set receive message descriptor 2 */
400         recv->must_be_ones = 0xFF;
401         recv->buffer_byte_count = MAX_RECEIVE_BUFFER_SIZE;
402
403         /* Set receive message descriptor 3 */
404         recv->must_be_zeros = 0;
405         recv->message_byte_count = 0;
406
407         ++recv; /* advance to the next receive descriptor. */
408     }
409
410     /*
411     * Set an unchanging pointer to the end of the receive
412     * message descriptor ring so that we know when to wrap around
413     * to as more and more messages are received (see lance_receive()).
414     */
415     maximum_receive_pointer = recv;
416 }
417
418
419 u_short
420 initialize_lance(recv, trans)
421 register RECV_MODE *recv;
422 register TRANS_MODE *trans;
423 {
424     if (set_up_control_and_status_registers() == FAILURE)
425         return(FAILURE);
426
427     set_up_initialization_block(recv, trans);
428
429     return(SUCCESS);
430 }
431
432
433 static u_short
434 set_up_control_and_status_registers()
435 {
436     /*
437     * The LANCE requires that the STOP bit of CS0 be set
438     * before accessing CS0, CS1, or CS2.
439     */
440     if (write_lance_car(SELECT_CS0, STOP_ACTIVITY) == FAILURE)
441         return(FAILURE);
442
443     /*
444     * The CPU requires that byte control and swap be set.
445     */
446     if (write_lance_car(SELECT_CS3, BYTE_CONTROL|BYTE_SWAP) == FAILURE)
447         return(FAILURE);
448
449     /*
450     * Load in the address of the initialization block.
451     */
452     if (write_lance_car(SELECT_CS2, (u_short) (0xFF & ((u_long) init_block_mem >> 16))) == FAILURE)
453         return(FAILURE);
454     if (write_lance_car(SELECT_CS1, (u_short) (0xFFFF & (u_long) init_block_mem)) == FAILURE)
455         return(FAILURE);
456
457     return(SUCCESS);
458 }
459
460
461 static void
462 set_up_initialization_block(recv, trans)
463 register RECV_MODE *recv;
464 register TRANS_MODE *trans;
465 {
466     register u_char *set_address;
467     register INIT_BLOCK *init_block;
468
469     extern ID_PROM_CPU id_prom;
470
471     init_block = (INIT_BLOCK *) init_block_mem;
472
473     /* Set up the correct operational modes */
474     init_block->promiscuous = 0;
475     init_block->reserved_1 = 0;
476     init_block->internal_loopback = 0;
477     init_block->disable_retry = 0;
478 }

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/lance.c

DATE 5/23/89 PAGE #
TIME 4:42:12 pm 5/21

```

LINE # SOURCE TEXT
481 init_block->force_collision = 0;
482 init_block->disable_crc_transmit = 0;
483 init_block->loopback = 0;
484 init_block->disable_transmit = 0;
485 init_block->disable_receive = 0;
486
487 /* Set up the 48-bits for the physical Ethernet address */
488 /* Note the byte sequence (see lance.h for details) */
489 const_address = (u_char *) id_prom.ethernet;
490 init_block->physical_address_1 = *const_address++;
491 init_block->physical_address_2 = *const_address++;
492 init_block->physical_address_3 = *const_address++;
493 init_block->physical_address_4 = *const_address++;
494 init_block->physical_address_5 = *const_address++;
495 init_block->physical_address_6 = *const_address++;
496
497 /* Set up the logical address filter */
498 /* Must always be zero (0) as we don't implement multi-casting */
499 init_block->logical_address_1 = 0;
500 init_block->logical_address_2 = 0;
501
502 /* Set up the receive descriptor ring pointer */
503 init_block->low_receive_ring_address = (u_short) recv;
504 init_block->high_receive_ring_address = (unsigned) recv >> 16;
505 init_block->receive_ring_length = POWER_OF_2_MAX_RECEIVE_BUFFERS;
506 init_block->reserved_2 = 0;
507
508 /* Set up the transmit descriptor ring pointer */
509 init_block->low_transmit_ring_address = (u_short) trans;
510 init_block->high_transmit_ring_address = (unsigned) trans >> 16;
511 init_block->transmit_ring_length = POWER_OF_2_MAX_TRANSMIT_BUFFERS;
512 init_block->reserved_3 = 0;
513
514
515
516 /* This is the fast access method to I/O CSRs
517 * The interrupt routine calls this to speed up access to the lance
518 * It reads CSRs then clears out bits and enables INTs
519 * From Lance.
520 */
521 u_short
522 isr_read_write_lance_car( value )
523 register u_short *value;
524 {
525     register u_short i;
526     register u_short *lance_data_register;
527     register u_short *lance_address_register;
528
529     register cpu_control_reg_struct *cpu_control_register= (cpu_control_reg_struct *) CPU_CONTROL_REG;
530
531     /*
532     * Initialize our local register variables.
533     */
534     lance_data_register = (u_short *) CPU_LANCE_DATA_REG;
535     lance_address_register = (u_short *) CPU_LANCE_ADDR_REG;
536
537     /*
538     * Don't change this code unless you carefully examine the assembler
539     * that is produced to ensure that the code will properly handle
540     * the hardware idiosyncrasies of the LANCE. Talk with Mark if
541     * you have any questions regarding the LANCE's functionality.
542     */
543
544     i = MAX_LANCE_ENABLE_POLL;
545     cpu_control_register->lance_enable = 0;
546     while (cpu_control_register->lance_busy == 1) {
547         if (--i) {
548             cpu_control_register->lance_enable = 1; /* Why? */
549             return(FAILURE);
550         }
551     }
552
553     *lance_address_register = SELECT_CSR;
554     *value = *lance_data_register;
555
556     *lance_data_register = *value | INTERRUPT_ENABLE;
557     cpu_control_register->lance_enable = 1;
558
559     return(SUCCESS);
560 }
561
562
563
564 u_short
565 write_lance_car(which_car, value)
566 register u_short which_car;
567 register u_short value;
568 {
569     register u_short i;
570     register u_short *lance_data_register;
571     register u_short *lance_address_register;
572     register u_short interrupts_were_enabled;
573
574     register cpu_control_reg_struct *cpu_control_register= (cpu_control_reg_struct *) CPU_CONTROL_REG;
575
576     /*
577     * Initialize our local register variables.
578     */
579     lance_data_register = (u_short *) CPU_LANCE_DATA_REG;
580     lance_address_register = (u_short *) CPU_LANCE_ADDR_REG;
581
582     /*
583     * Don't change this code unless you carefully examine the assembler
584     * that is produced to ensure that the code will properly handle
585     * the hardware idiosyncrasies of the LANCE. Talk with Mark if
586     * you have any questions regarding the LANCE's functionality.
587     */
588
589     /*
590     * MUST disable interrupts during this time as the data that
591     * gets written depends upon which data port is selected by
592     * the CPU_LANCE_ADDR_REG, among other things.
593     */
594     interrupts_were_enabled = cpu_control_register->global_intr_ens;
595     CPU_DISABLE_INTERRUPTS;
596
597     /*
598     * Inhibit the LANCE from doing anything while we get at its CSR.
599     */
600

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lance.c | DATE 5/23/89 | PAGE # 6/22 |
|--|--|------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 601 | * Then test several times for the LANCE being busy. When its | | | |
| 602 | * free, look at the value in the LANCE's data register. | | | |
| 603 | * The address port must be written before the data port. | | | |
| 604 | * Note that we reset the LANCE's address port to reference | | | |
| 605 | * CSIO. See the comment above for this "feature's" motivation. | | | |
| 606 | * Enable the LANCE after we're done accessing it's CSR. | | | |
| 607 | /* | | | |
| 608 | | | | |
| 609 | i = MAX_LANCE_ENABLE_POLLS; | | | |
| 610 | cpu_control_register->lance_enable = 0; | | | |
| 611 | while (cpu_control_register->lance_busy == 1) { | | | |
| 612 | if (--i) { | | | |
| 613 | cpu_control_register->lance_enable = 1; /* Why? */ | | | |
| 614 | return(FAILURE); | | | |
| 615 | } | | | |
| 616 | | | | |
| 617 | } | | | |
| 618 | *lance_address_register = which_car; | | | |
| 619 | *lance_data_register = value; | | | |
| 620 | cpu_control_register->lance_enable = 1; | | | |
| 621 | | | | |
| 622 | /* | | | |
| 623 | * Turn interrupts back on, only if they were enabled before. | | | |
| 624 | */ | | | |
| 625 | if (interrupts_were_enabled) | | | |
| 626 | CPU_ENABLE_INTERRUPTS; | | | |
| 627 | | | | |
| 628 | return(SUCCESS); | | | |
| 629 | | | | |
| 630 | } | | | |
| 631 | | | | |
| 632 | | | | |
| 633 | u_short | | | |
| 634 | read_lance_car(which_car, value) | | | |
| 635 | register u_short which_car; | | | |
| 636 | register u_short *value; | | | |
| 637 | { | | | |
| 638 | register u_short i; | | | |
| 639 | register u_short *lance_data_register; | | | |
| 640 | register u_short *lance_address_register; | | | |
| 641 | register u_short interrupts_were_enabled; | | | |
| 642 | | | | |
| 643 | register cpu_control_reg_struct *cpu_control_register= (cpu_control_reg_struct *) CPU_CONTROL_REG; | | | |
| 644 | | | | |
| 645 | | | | |
| 646 | /* | | | |
| 647 | * Initialize our local register variables. | | | |
| 648 | */ | | | |
| 649 | lance_data_register = (u_short *) CPU_LANCE_DATA_REG; | | | |
| 650 | lance_address_register = (u_short *) CPU_LANCE_ADDR_REG; | | | |
| 651 | | | | |
| 652 | /* | | | |
| 653 | * Don't change this code unless you carefully examine the assembler | | | |
| 654 | * that is produced to ensure that the code will properly handle | | | |
| 655 | * the hardware idiosyncrasies of the LANCE. Talk with Mark if | | | |
| 656 | * you have any questions regarding the LANCE's functionality. | | | |
| 657 | */ | | | |
| 658 | | | | |
| 659 | /* | | | |
| 660 | * MUST disable interrupts during this time as the data that | | | |
| 661 | * gets written depends upon which data port is selected by | | | |
| 662 | * the CPU_LANCE_ADDR_REG, among other things. | | | |
| 663 | */ | | | |
| 664 | interrupts_were_enabled = cpu_control_register->global_intr_ena; | | | |
| 665 | CPU_DISABLE_INTERRUPTS; | | | |
| 666 | | | | |
| 667 | /* | | | |
| 668 | * Inhibit the LANCE from doing anything while we get at its CSR. | | | |
| 669 | */ | | | |
| 670 | * Then test several times for the LANCE being busy. When its | | | |
| 671 | * free, look at the value in the LANCE's data register. | | | |
| 672 | * The address port must be written before the data port. | | | |
| 673 | * Note that we reset the LANCE's address port to reference | | | |
| 674 | * CSIO. See the comment above for this "feature's" motivation. | | | |
| 675 | * Enable the LANCE after we're done accessing it's CSR. | | | |
| 676 | */ | | | |
| 677 | | | | |
| 678 | | | | |
| 679 | i = MAX_LANCE_ENABLE_POLLS; | | | |
| 680 | cpu_control_register->lance_enable = 0; | | | |
| 681 | while (cpu_control_register->lance_busy == 1) { | | | |
| 682 | if (--i) { | | | |
| 683 | cpu_control_register->lance_enable = 1; /* Why? */ | | | |
| 684 | return(FAILURE); | | | |
| 685 | } | | | |
| 686 | | | | |
| 687 | *lance_address_register = which_car; | | | |
| 688 | *value = *lance_data_register; | | | |
| 689 | cpu_control_register->lance_enable = 1; | | | |
| 690 | | | | |
| 691 | /* | | | |
| 692 | * Turn interrupts back on, only if they were enabled before. | | | |
| 693 | */ | | | |
| 694 | if (interrupts_were_enabled) | | | |
| 695 | CPU_ENABLE_INTERRUPTS; | | | |
| 696 | | | | |
| 697 | return(SUCCESS); | | | |
| 698 | | | | |
| 699 | } | | | |
| 700 | | | | |
| 701 | | | | |
| 702 | int | | | |
| 703 | sendto(id, message, length, flag, sock, sock_size) | | | |
| 704 | int id, /* ignored */ | | | |
| 705 | register char *message; | | | |
| 706 | int length; | | | |
| 707 | int flag; | | | |
| 708 | register SOCKET_ADDRESS *sock; | | | |
| 709 | int sock_size, /* ignored */ | | | |
| 710 | { | | | |
| 711 | u_long error; | | | |
| 712 | | | | |
| 713 | register ETHERNET_HEADER *enet_header; | | | |
| 714 | register IP_HEADER *ip_header; | | | |
| 715 | register UDP_HEADER *udp_header; | | | |
| 716 | register INIT_BLOCK *init_block; | | | |
| 717 | | | | |
| 718 | extern u_short compute_ip_checksum(); | | | |
| 719 | | | | |
| 720 | | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/lance.c

DATE 5/23/89
TIME 4:42:12 pm

PAGE #
7/23

```

LINE #
721 /* Initialize the network headers on the outgoing message.
722 */
723 enet_header = (ETHERNET_HEADER *) message;
724
725 /*
726 * Copy the Ethernet source address from the LANCE's
727 * initialization block to the outgoing message buffer.
728 */
729 init_block = (INIT_BLOCK *) init_block_ptr;
730 enet_header->source[0] = init_block->physical_address_1;
731 enet_header->source[1] = init_block->physical_address_2;
732 enet_header->source[2] = init_block->physical_address_3;
733 enet_header->source[3] = init_block->physical_address_4;
734 enet_header->source[4] = init_block->physical_address_5;
735 enet_header->source[5] = init_block->physical_address_6;
736
737 /*
738 * Copy the Ethernet destination address from the
739 * message socket to the outgoing message buffer.
740 */
741 enet_header->destination[0] = sock->destination[0];
742 enet_header->destination[1] = sock->destination[1];
743 enet_header->destination[2] = sock->destination[2];
744 enet_header->destination[3] = sock->destination[3];
745 enet_header->destination[4] = sock->destination[4];
746 enet_header->destination[5] = sock->destination[5];
747
748 /*
749 * Copy the packet type from the message socket to the
750 * outgoing message buffer.
751 */
752 enet_header->type = sock->packet_type;
753
754 if ((flag & UDP_IP_PACKET) == UDP_IP_PACKET) {
755     ip_header = (IP_HEADER *) (message + sizeof(ETHERNET_HEADER));
756     udp_header = (UDP_HEADER *) (message + sizeof(ETHERNET_HEADER) + sizeof(IP_HEADER));
757
758     ip_header->version_and_header_length = 0x45; /* IP */
759     ip_header->type_of_service = 0x00; /* not used */
760     ip_header->total_length = length + sizeof(IP_HEADER) + sizeof(UDP_HEADER); /* length of packet w/ headers */
761     ip_header->identification = 0x0000; /* not used */
762     ip_header->fragment_offset = 0x0000; /* not used */
763     ip_header->time_to_live = 0x1f; /* 255 seconds */
764     ip_header->protocol = 0x11; /* UDP */
765     ip_header->checksum = 0x0000; /* computed later */
766
767     if (((flag & ALIVE_PACKET) == ALIVE_PACKET) ||
768         ((flag & DEATH_PACKET) == DEATH_PACKET)) {
769         ip_header->source_address = alive_ipnet;
770         ip_header->destination_address = modeler_ipnet;
771     } else {
772         ip_header->source_address = sock->address;
773         ip_header->checksum = compute_ip_checksum((u_short *) ip_header);
774     }
775
776     if ((flag & ALIVE_PACKET) == ALIVE_PACKET) {
777         udp_header->source_port = MODELERS_ALIVE_PORT;
778     } else if ((flag & DEATH_PACKET) == DEATH_PACKET) {
779         udp_header->source_port = MODELERS_DEATH_PORT;
780     } else {
781         udp_header->source_port = MODELERS_ETHERNET_PORT;
782     }
783     udp_header->destination_port = sock->port;
784     udp_header->length = (u_short) (length + sizeof(UDP_HEADER));
785     udp_header->checksum = 0x0000; /* Not used */
786
787     if (lance_transmit(message, (u_short) (length + sizeof(ETHERNET_HEADER) + sizeof(IP_HEADER) + sizeof(UDP_HEADER)), &error) != SUCCESS) {
788         return(-error); /* Must return negative number to indicate error. */
789     }
790 }
791
792 else { /* RAW PACKET */
793     if (lance_transmit(message, (u_short) (length + sizeof(ETHERNET_HEADER)), &error) != SUCCESS) {
794         return(-error); /* Must return negative number to indicate error. */
795     }
796 }
797
798 return(length);
799
800 static u_short
801 compute_ip_checksum (a)
802 register u_short *a;
803 {
804     /*
805      * This function computes the IP header checksum. The header is passed in as
806      * an array of unsigned shorts. This function should be called only after all
807      * of the bytes have been appropriately swapped. The resulting checksum must
808      * NOT be byte swapped. (Stolen from RI's VMS network code.)
809      */
810     register u_long sum;
811
812     /* Sum up the 10 words (20 bytes) of the ip header. */
813     sum = *a;
814     sum += **a;
815     sum += **a;
816     sum += **a;
817     sum += **a;
818     sum += **a;
819     sum += **a;
820     sum += **a;
821     sum += **a;
822     sum += **a;
823
824     sum += sum >> 16; /* add in the carry */
825     return((u_short) ~sum); /* returns the one's complement of the checksum */
826
827 int
828 recvfrom (fd, message, length, option, from, from_len)
829 int fd;
830 char *message;
831 int length;
832 int option;
833 SOCKET_ADDRESS *from;
834 int *from_len;
835 {
836     u_long error;
837     u_short error_code, actual_length;
838     int err;
839 }

```


Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/lance.c

DATE 5/23/89
TIME 4:42:12 pm

PAGE #
8/24

```

LINE #      SOURCE TEXT
840      register      ETHERNET_HEADER *enet_header;
841      register      RCVR_PKT *rcvr;
842      register      IP_HEADER *ip_header;
843      register      UDP_HEADER *udp_header;
844      register      struct ethernet_arp_packet *packet;
845
846      while( 1 )
847      {
848          /*
849          ** wait for a message from host
850          */
851          CPU_DISABLE_INTERRUPTS;
852          while( !extra_rcv_desc_count )
853          {
854              CPU_ENABLE_INTERRUPTS;
855              if( option == RCVR_PKT_TIMEOUT )
856                  ac_spnd( rcv_lance_pkt_semaphore, SECOND * TIMEOUT, &err );
857              else
858                  ac_spnd( rcv_lance_pkt_semaphore, 0, &err );
859              if( err )
860                  return( -1 );
861          }
862          CPU_ENABLE_INTERRUPTS;
863          /*
864          ** there is something from a host
865          */
866          rcvr = extra_rcv_desc_buffers[ out_extra_rcv_desc_ptr ];
867
868          /*
869          ** get message
870          */
871          if ( lance_receive(message, (u_short)length,
872                          &actual_length, rcvr, option, &error) != SUCCESS)
873          {
874              if( option != MSG_PEEK )
875              {
876                  CPU_DISABLE_INTERRUPTS;
877                  extra_rcv_desc_count--;
878                  out_extra_rcv_desc_ptr++;
879                  out_extra_rcv_desc_ptr = ( MAX_USERS - 1 );
880                  p_lm_rcv_buffers_extra[ ++available_extra_buffer ]
881                      = (RCVR_PKT *)((rcvr->high_buffer_address << 16)
882                      | rcvr->low_buffer_address);
883                  if( too_many_buffers == 1 )
884                      lance_isr_receive( &error_code );
885                  CPU_ENABLE_INTERRUPTS;
886              }
887
888              /* Next returns negative number to indicate error. */
889              if (error <= 0)
890                  return(-1);
891              else return(-error);
892          }
893          if( option != MSG_PEEK )
894          {
895              CPU_DISABLE_INTERRUPTS;
896              extra_rcv_desc_count--;
897              out_extra_rcv_desc_ptr++;
898              out_extra_rcv_desc_ptr = ( MAX_USERS - 1 );
899              p_lm_rcv_buffers_extra[ ++available_extra_buffer ]
900                  = (RCVR_PKT *)((rcvr->high_buffer_address << 16)
901                  | rcvr->low_buffer_address);
902              if( too_many_buffers == 1 )
903                  lance_isr_receive( &error_code );
904              CPU_ENABLE_INTERRUPTS;
905          }
906
907          udp_header = (UDP_HEADER *) (message+sizeof(ETHERNET_HEADER)+sizeof(IP_HEADER));
908          ip_header = (IP_HEADER *) (message + sizeof(ETHERNET_HEADER));
909
910          packet = (struct ethernet_arp_packet *) message;
911          if ( packet->arp.arp_opcode == ARP_REPLY )
912          {
913              return(sizeof(struct ethernet_arp_packet));
914          }
915          enet_header = (ETHERNET_HEADER *) message;
916          from->packet_type = enet_header->type;
917          from->destination[0] = enet_header->source[0];
918          from->destination[1] = enet_header->source[1];
919          from->destination[2] = enet_header->source[2];
920          from->destination[3] = enet_header->source[3];
921          from->destination[4] = enet_header->source[4];
922          from->destination[5] = enet_header->source[5];
923
924          from->port = udp_header->source_port;
925          from->address = ip_header->source_address;
926          from->family = AF_INET;
927
928          modeler_inet_address = ip_header->destination_address;
929          from_len = SOCK_SIZE;
930
931          actual_length = ip_header->total_length - (sizeof( IP_HEADER ) + sizeof( UDP_HEADER )); /* length of packet w/o headers */
932
933          /*
934          ** if we get a packet with funny insides just continue.
935          */
936          if( actual_length > MAX_RECEIVE_BUFFER_SIZE || actual_length == 0 )
937              continue;
938
939          return(actual_length);
940      }
941
942      u_short
943      lance_transmit(message, length, error)
944      register      char *message;
945      register      u_short length;
946      register      u_long *error;
947      {
948          register      TRANS_PKT *trans;
949          register      u_short interrupts_were_enabled;
950
951          /*
952          ** Start out with no errors.
953          */
954          *error = 0;
955      }

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lance.c | DATE 5/23/89 | PAGE # 9/25 |
|--|--|------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 960 | interrupts_were_enabled = | | | |
| 961 | ((cps_control_reg_struct *) CPU_CONTROL_REG->global_intr_ess, | | | |
| 962 | CPU_DISABLE_INTERRUPTS, | | | |
| 963 | /* | | | |
| 964 | /* Local register pointer. | | | |
| 965 | */ | | | |
| 966 | trans = current_transmit_pointer; | | | |
| 967 | /* | | | |
| 968 | /* Set our, as opposed to the LANCE's, notion of what the current transmit | | | |
| 969 | message descriptor is. Actually, this is the next transmit descriptor | | | |
| 970 | to be used, but I think the code is clearer if it's name is "current". | | | |
| 971 | /* | | | |
| 972 | /* The transmit message descriptors are in a ring structure which causes | | | |
| 973 | us to have to wrap around to the beginning when the end of the list | | | |
| 974 | is reached. | | | |
| 975 | */ | | | |
| 976 | if (++current_transmit_pointer >= maximum_transmit_pointer) { | | | |
| 977 | current_transmit_pointer = minimum_transmit_pointer; | | | |
| 978 | } | | | |
| 979 | if (interrupts_were_enabled) | | | |
| 980 | CPU_ENABLE_INTERRUPTS, | | | |
| 981 | /* | | | |
| 982 | if (trans->own_buffer == LANCE_BUFFER_OWNERSHIP) | | | |
| 983 | return(FAILURE); | | | |
| 984 | /* | | | |
| 985 | /* Set up the address of the message that is being sent, indicate that | | | |
| 986 | this is both the start and the end of this packet, set the packet | | | |
| 987 | byte count, finally - note that this must be done last - the | | | |
| 988 | buffer is turned over to the LANCE for transmission. | | | |
| 989 | */ | | | |
| 990 | trans->low_buffer_address = (u_short) message; | | | |
| 991 | trans->high_buffer_address = (unsigned) message >> 16; | | | |
| 992 | trans->start_of_packet = 1; | | | |
| 993 | trans->end_of_packet = 1; | | | |
| 994 | /* | | | |
| 995 | /* Most hosts will not like very short packets | | | |
| 996 | /* 64 is a minimum for SUN's | | | |
| 997 | /* | | | |
| 998 | if (allow_runt_packets == FALSE) { | | | |
| 999 | if (length < 64) { | | | |
| 1000 | length = 64; | | | |
| 1001 | } | | | |
| 1002 | } | | | |
| 1003 | trans->buffer_byte_count = (unsigned) (1 + "length); | | | |
| 1004 | trans->own_buffer = LANCE_BUFFER_OWNERSHIP; | | | |
| 1005 | /* | | | |
| 1006 | /* Poke the LANCE to let it know that we have a message for it to send. | | | |
| 1007 | /* Otherwise, it would wait until the polling time interval had elapsed. | | | |
| 1008 | /* Note that we must set the interrupt enable bit ON every time the | | | |
| 1009 | transmit demand bit is set, because for some unknown reason, interrupts | | | |
| 1010 | are disabled whenever we set transmit demand. | | | |
| 1011 | */ | | | |
| 1012 | if (write_lance_csr(SELECT_CSR0, TRANSMIT_DEMAND INTERRUPT_ENABLE) == FAILURE) { | | | |
| 1013 | error = FAILURE_TO_WRITE_TO_LANCE_CSR; | | | |
| 1014 | return(FAILURE); | | | |
| 1015 | } | | | |
| 1016 | return(SUCCESS); | | | |
| 1017 | /* | | | |
| 1018 | static u_short | | | |
| 1019 | lance_receive(buffer, max_length, actual_length, rcv, options, error) | | | |
| 1020 | register char *buffer; | | | |
| 1021 | register u_short max_length; | | | |
| 1022 | register u_short actual_length; | | | |
| 1023 | register u_long error; | | | |
| 1024 | register int options; | | | |
| 1025 | register RECV_MSG *rcv; | | | |
| 1026 | { | | | |
| 1027 | register char *rcv_buffer; | | | |
| 1028 | /* | | | |
| 1029 | /* Start out with no errors. | | | |
| 1030 | */ | | | |
| 1031 | error = 0; | | | |
| 1032 | /* | | | |
| 1033 | /* This implementation of the LANCE code requires that the entire | | | |
| 1034 | message fit within a single receive message buffer. Check that | | | |
| 1035 | the LANCE thinks that this has happened. | | | |
| 1036 | */ | | | |
| 1037 | if (rcv->start_of_packet != 1) | | | |
| 1038 | error = PACKET_OVERFLOW_ERROR; | | | |
| 1039 | if (rcv->end_of_packet != 1) | | | |
| 1040 | error = PACKET_OVERFLOW_ERROR; | | | |
| 1041 | /* | | | |
| 1042 | if (error != 0) | | | |
| 1043 | return(FAILURE); | | | |
| 1044 | /* | | | |
| 1045 | /* Looks like we have a good message. Let's process and return it | | | |
| 1046 | to the anxious user. | | | |
| 1047 | */ | | | |
| 1048 | /* | | | |
| 1049 | /* Check to see if the size of the received message is small enough | | | |
| 1050 | to fit into the buffer that the user passed us. The number of | | | |
| 1051 | bytes that the LANCE deals with is 4 more than one would expect | | | |
| 1052 | because of the 4 byte CRC. Adjust the actual length as indicated. | | | |
| 1053 | */ | | | |
| 1054 | actual_length = rcv->message_byte_count-4; | | | |
| 1055 | if (options != MSG_PEEK) | | | |
| 1056 | { | | | |
| 1057 | if (actual_length > max_length) | | | |
| 1058 | { | | | |
| 1059 | error = BUFFER_TOO_SMALL; | | | |
| 1060 | return(FAILURE); | | | |
| 1061 | } | | | |
| 1062 | } | | | |
| 1063 | /* | | | |
| 1064 | /* Copy the received buffer to the user's buffer. | | | |
| 1065 | */ | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lance.c | DATE 5/23/89 TIME 4:42:12 pm | PAGE # 10/26 |
|--|---|------------------------------|---------------------------------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 1080 | recv_buffer = (char *) ((recv->high_buffer_address << 16) recv->low_buffer_address); | | | |
| 1081 | /* | | | |
| 1082 | * If the packet is not a broadcast, copy the data from the LANCE's buffer | | | |
| 1083 | * to the calling routine's buffer. Broadcast packets are indicated by | | | |
| 1084 | * a returned length of 0 - no data is copied. | | | |
| 1085 | */ | | | |
| 1086 | if(option != MSG_PEEK) | | | |
| 1087 | memory_copy((u_long *)buffer, (u_long *)recv_buffer, (short)*actual_length); | | | |
| 1088 | else | | | |
| 1089 | memory_copy((u_long *)buffer, (u_long *)recv_buffer, (short)max_length); | | | |
| 1090 | /* | | | |
| 1091 | * We must adjust the length of the actual message to compensate for the | | | |
| 1092 | * calling routines lack of knowledge of the Ethernet header. | | | |
| 1093 | */ | | | |
| 1094 | *actual_length -- sizeof(ETHERNET_HEADER); | | | |
| 1095 | | | | |
| 1096 | return(SUCCESS); | | | |
| 1097 | | | | |
| 1098 | } | | | |
| 1099 | | | | |
| 1100 | /* | | | |
| 1101 | * The code has been changed so that it is not possible for lance_isr_receive() | | | |
| 1102 | * to return any failures. If the calling routine wants to see if "warnings" | | | |
| 1103 | * occurred, they can look at the returned "error". | | | |
| 1104 | */ | | | |
| 1105 | static void | | | |
| 1106 | lance_isr_receive(error) | | | |
| 1107 | register u_short *error; | | | |
| 1108 | { | | | |
| 1109 | register char *recv_buffer; | | | |
| 1110 | | | | |
| 1111 | register RECV_BUF *recv; | | | |
| 1112 | register IP_HEADER *ip_header; | | | |
| 1113 | register UDP_HEADER *udp_header; | | | |
| 1114 | register LM_HEADER *lm_header; | | | |
| 1115 | register ICMP_HEADER *icmp_pkt; | | | |
| 1116 | | | | |
| 1117 | extern char lm_valid_address; | | | |
| 1118 | extern void process_other_stuff(); | | | |
| 1119 | extern void process_are_you_alive(); | | | |
| 1120 | | | | |
| 1121 | /* | | | |
| 1122 | * Start out with no errors. | | | |
| 1123 | */ | | | |
| 1124 | *error = 0; | | | |
| 1125 | | | | |
| 1126 | /* | | | |
| 1127 | * Note: in the following code we must wait until the last possible | | | |
| 1128 | * moment to turn the buffer back over to the LANCE. Therefore, don't | | | |
| 1129 | * execute the following line of code until you have to: | | | |
| 1130 | */ | | | |
| 1131 | /* recv->own_buffer = LANCE_BUFFER_OWNERSHIP; */ | | | |
| 1132 | | | | |
| 1133 | | | | |
| 1134 | | | | |
| 1135 | | | | |
| 1136 | while (current_receive_pointer->own_buffer == HOST_BUFFER_OWNERSHIP) { | | | |
| 1137 | /* | | | |
| 1138 | * Local register pointer. | | | |
| 1139 | */ | | | |
| 1140 | recv = current_receive_pointer; | | | |
| 1141 | /* | | | |
| 1142 | * Check for errors that occurred during reception. | | | |
| 1143 | */ | | | |
| 1144 | if (recv->error_summary == 1) { | | | |
| 1145 | if (recv->framing_error == 1) { | | | |
| 1146 | *error = FRAMING_ERROR; | | | |
| 1147 | /* output_routine("FE"); /* Framing Error */ | | | |
| 1148 | } | | | |
| 1149 | if (recv->overflow_error == 1) { | | | |
| 1150 | *error = OVERFLOW_ERROR; | | | |
| 1151 | /* output_routine("OF"); /* Overflow error */ | | | |
| 1152 | } | | | |
| 1153 | if (recv->crc_error == 1) { | | | |
| 1154 | *error = CRC_ERROR; | | | |
| 1155 | /* output_routine("CRC"); /* CRC error */ | | | |
| 1156 | } | | | |
| 1157 | if (recv->buffer_error == 1) { | | | |
| 1158 | *error = BUFFER_ERROR; | | | |
| 1159 | /* output_routine("BU"); /* Buffer error */ | | | |
| 1160 | } | | | |
| 1161 | /* Turn the buffer back over to the LANCE. */ | | | |
| 1162 | recv->own_buffer = LANCE_BUFFER_OWNERSHIP; | | | |
| 1163 | | | | |
| 1164 | if (++current_receive_pointer >= maximum_receive_pointer) | | | |
| 1165 | current_receive_pointer = minimum_receive_pointer; | | | |
| 1166 | | | | |
| 1167 | continue; | | | |
| 1168 | } | | | |
| 1169 | | | | |
| 1170 | /* Get the address of the buffer. */ | | | |
| 1171 | recv_buffer = (char *) ((recv->high_buffer_address << 16) recv->low_buffer_address); | | | |
| 1172 | | | | |
| 1173 | /* Set up pointers to the UDP and IP parts of the buffer. */ | | | |
| 1174 | ip_header = (IP_HEADER *) (recv_buffer + sizeof(ETHERNET_HEADER)); | | | |
| 1175 | udp_header = (UDP_HEADER *) (recv_buffer + sizeof(ETHERNET_HEADER) + sizeof(IP_HEADER)); | | | |
| 1176 | lm_header = (LM_HEADER *) (recv_buffer + sizeof(ETHERNET_HEADER) + sizeof(IP_HEADER) + sizeof(UDP_HEADER)); | | | |
| 1177 | | | | |
| 1178 | /* Reject broadcast, non-UDP, and incorrect port packets. */ | | | |
| 1179 | | | | |
| 1180 | if (((u_long *)recv_buffer != 0xffffffff) && ((u_short *) (recv_buffer+4) != 0xffff)) { | | | |
| 1181 | if (((u_short *) (recv_buffer+12) == 0x0800) { /* type is IP */ | | | |
| 1182 | | | | |
| 1183 | if (ip_header->protocol == 0x11) { | | | |
| 1184 | switch(udp_header->destination_port) { | | | |
| 1185 | case MODELS_ARE_YOU_ALIVE_PORT: | | | |
| 1186 | process_are_you_alive(recv_buffer, | | | |
| 1187 | ip_header, udp_header, lm_header, | | | |
| 1188 | ALIVE_PACKET); | | | |
| 1189 | break; | | | |
| 1190 | case MODELS_DEATH_PORT: | | | |
| 1191 | /* respond as if are you alive */ | | | |
| 1192 | process_are_you_alive(recv_buffer, | | | |
| 1193 | ip_header, udp_header, lm_header, | | | |
| 1194 | DEATH_PACKET); | | | |
| 1195 | output_routine("REMOTE RESET"); | | | |
| 1196 | /* I think I'm gonna kill myself */ | | | |
| 1197 | reset_cpu(REMOTE_RESET); | | | |
| 1198 | | | | |
| 1199 | | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lance.c | DATE 5/23/89 TIME 4:42:12 pm | PAGE # 11/27 |
|--|---|------------------------------|---------------------------------|-----------------|
| SOURCE TEXT | | | | |
| LINE # | SOURCE TEXT | | | |
| 1200 | break; | | | |
| 1201 | case MODELER_ETHERNET_PORT: | | | |
| 1202 | case MODELER_OUT_OF_BAND_PORT: | | | |
| 1203 | process_modeler_stuff(recv); | | | |
| 1204 | break; | | | |
| 1205 | } | | | |
| 1206 | } | | | |
| 1207 | else | | | |
| 1208 | { | | | |
| 1209 | /* | | | |
| 1210 | ** we have a ICMP | | | |
| 1211 | */ | | | |
| 1212 | if(ip_header->protocol == 0x1) | | | |
| 1213 | { | | | |
| 1214 | icmp_pkt = (ICMP_HEADER *)udp_header; | | | |
| 1215 | /* | | | |
| 1216 | ** check for echo request | | | |
| 1217 | */ | | | |
| 1218 | if(icmp_pkt->type == 8) | | | |
| 1219 | { | | | |
| 1220 | send_icmp_reply(recv_buffer); | | | |
| 1221 | } | | | |
| 1222 | } | | | |
| 1223 | } | | | |
| 1224 | } | | | |
| 1225 | } else if (*(u_short *) (recv_buffer+12) == ARP_ARN) { /* type is ARP */ | | | |
| 1226 | { | | | |
| 1227 | struct ethernet_arp_packet *packet; | | | |
| 1228 | long error; | | | |
| 1229 | packet = (struct ethernet_arp_packet *) recv_buffer; | | | |
| 1230 | if (packet->arp.arp_opcode == ARP_REPLY && accept_arp_reply) { | | | |
| 1231 | { | | | |
| 1232 | /* We have received a reply to our ARP | | | |
| 1233 | request. Pass it to AI code | | | |
| 1234 | */ | | | |
| 1235 | process_modeler_stuff(recv); | | | |
| 1236 | } | | | |
| 1237 | if (packet->arp.arp_opcode == ARP_REQUEST) { | | | |
| 1238 | { | | | |
| 1239 | /* This is an ARP request directed at the | | | |
| 1240 | modeler. Here is where we should | | | |
| 1241 | initiate a response. | | | |
| 1242 | */ | | | |
| 1243 | if(!in_valid_bvaran == TRUE) | | | |
| 1244 | { | | | |
| 1245 | send_arp_reply ((struct ethernet_arp_packet *) recv_buffer, | | | |
| 1246 | (u_char *) id_prom.ethernet, | | | |
| 1247 | boot.modeler_internet_address, | | | |
| 1248 | packet->arp.arp_source_hardware_address, | | | |
| 1249 | packet->arp.arp_source_protocol_address); | | | |
| 1250 | } | | | |
| 1251 | } | | | |
| 1252 | } | | | |
| 1253 | } | | | |
| 1254 | } | | | |
| 1255 | } else { | | | |
| 1256 | /* We have a broadcast packet which might be of interest */ | | | |
| 1257 | { | | | |
| 1258 | struct ethernet_arp_packet *packet; | | | |
| 1259 | packet = (struct ethernet_arp_packet *) recv_buffer; | | | |
| 1260 | if (packet->arp.arp_opcode == ARP_REQUEST) { | | | |
| 1261 | { | | | |
| 1262 | if (packet->arp.arp_target_protocol_address == boot.modeler_internet_address) { | | | |
| 1263 | { | | | |
| 1264 | /* This is a broadcasted ARP request for | | | |
| 1265 | this modeler. Here is where we should | | | |
| 1266 | initiate a response. | | | |
| 1267 | */ | | | |
| 1268 | if(!in_valid_bvaran == TRUE) | | | |
| 1269 | { | | | |
| 1270 | send_arp_reply ((struct ethernet_arp_packet *) recv_buffer, | | | |
| 1271 | (u_char *) id_prom.ethernet, | | | |
| 1272 | boot.modeler_internet_address, | | | |
| 1273 | packet->arp.arp_source_hardware_address, | | | |
| 1274 | packet->arp.arp_source_protocol_address); | | | |
| 1275 | } | | | |
| 1276 | } | | | |
| 1277 | } | | | |
| 1278 | } | | | |
| 1279 | } | | | |
| 1280 | too_many_buffers = 0; | | | |
| 1281 | /* | | | |
| 1282 | Turn the buffer back over to the LANCE. | | | |
| 1283 | */ | | | |
| 1284 | recv->own_buffer = LANCE_BUFFER_OWNERSHIP; | | | |
| 1285 | /* | | | |
| 1286 | Set our, as opposed to the LANCE's, notion of what the current receive | | | |
| 1287 | message descriptor is. Actually, this is the next receive descriptor | | | |
| 1288 | to be used, but I think the code is clearer if it's name is "current". | | | |
| 1289 | */ | | | |
| 1290 | /* The receive message descriptors are in a ring structure which causes | | | |
| 1291 | us to have to wrap around to the beginning when the end of the list | | | |
| 1292 | is reached. | | | |
| 1293 | */ | | | |
| 1294 | if (++current_receive_pointer >= maximum_receive_pointer) | | | |
| 1295 | current_receive_pointer = minimum_receive_pointer; | | | |
| 1296 | } | | | |
| 1297 | } | | | |
| 1298 | } | | | |
| 1299 | } | | | |
| 1300 | } | | | |
| 1301 | static void | | | |
| 1302 | process_are_you_alive(message, ip_header, udp_header, lm_header, flag) | | | |
| 1303 | { | | | |
| 1304 | register char *message; | | | |
| 1305 | register IP_HEADER *ip_header; | | | |
| 1306 | register UDP_HEADER *udp_header; | | | |
| 1307 | register LM_HEADER *lm_header; | | | |
| 1308 | { | | | |
| 1309 | SOCKET_ADDRESS from; | | | |
| 1310 | register ETHERNET_HEADER *enet_header; | | | |
| 1311 | long *lptr = (long *) (lm_header + 1); | | | |
| 1312 | long cmd; | | | |
| 1313 | long size; | | | |
| 1314 | char *buffer; | | | |
| 1315 | { | | | |
| 1316 | extern char modeler_state; | | | |
| 1317 | { | | | |
| 1318 | cmd = *lptr++; | | | |
| 1319 | size = *lptr++; | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lance.c | DATE 5/23/89 | PAGE # 12/28 |
|--|--|------------------------------|-----------------|-----------------|
| LINE # | SOURCE TEXT | | | |
| 1320 | buffer = (char *)lptr; | | | |
| 1321 | | | | |
| 1322 | reset_header = (ETHERNET_HEADER *) message; | | | |
| 1323 | | | | |
| 1324 | from.packet_type = reset_header->type; | | | |
| 1325 | from.destination[0] = reset_header->source[0]; | | | |
| 1326 | from.destination[1] = reset_header->source[1]; | | | |
| 1327 | from.destination[2] = reset_header->source[2]; | | | |
| 1328 | from.destination[3] = reset_header->source[3]; | | | |
| 1329 | from.destination[4] = reset_header->source[4]; | | | |
| 1330 | from.destination[5] = reset_header->source[5]; | | | |
| 1331 | | | | |
| 1332 | from.family = AF_INET; | | | |
| 1333 | from.port = udp_header->source_port; | | | |
| 1334 | from.address = ip_header->source_address; | | | |
| 1335 | | | | |
| 1336 | alive_inet = ip_header->destination_address; | | | |
| 1337 | | | | |
| 1338 | lm_header->byte_count = sizeof(LM_HEADER) + 4; /* Choose to send 4 bytes. */ | | | |
| 1339 | /* All other lm_header fields are already properly set. */ | | | |
| 1340 | | | | |
| 1341 | /* NOTE: update the version when either "unknown" changes to a known. */ | | | |
| 1342 | | | | |
| 1343 | /* At the end of the header, put a version and the desired return data. */ | | | |
| 1344 | *(u_char *)((char *)lm_header+sizeof(LM_HEADER)+0) = (u_char) 0x1; /* version */ | | | |
| 1345 | *(u_char *)((char *)lm_header+sizeof(LM_HEADER)+1) = (u_char) modeler_state; | | | |
| 1346 | *(u_char *)((char *)lm_header+sizeof(LM_HEADER)+2) = (u_long) 0x0; /* unknown */ | | | |
| 1347 | *(u_char *)((char *)lm_header+sizeof(LM_HEADER)+3) = (u_long) 0x0; /* unknown */ | | | |
| 1348 | | | | |
| 1349 | (void) sendto(0, message, (int)lm_header->byte_count, | | | |
| 1350 | UDP_IP_PACKET flag, (struct sockaddr *)&from, SOCK_SIZE); | | | |
| 1351 | | | | |
| 1352 | switch (cmd) { | | | |
| 1353 | case AYA_INQUIRE: | | | |
| 1354 | break; | | | |
| 1355 | case AYA_KILL: | | | |
| 1356 | outputRoutine("sREMOTE RESET\n"); | | | |
| 1357 | /* I think I'm gonna kill myself */ | | | |
| 1358 | reset_cpu(REMOTE_RESET); | | | |
| 1359 | break; | | | |
| 1360 | case AYA_INTERRUPT: | | | |
| 1361 | lm_intr_requested = 1; | | | |
| 1362 | break; | | | |
| 1363 | } | | | |
| 1364 | | | | |
| 1365 | | | | |
| 1366 | | | | |
| 1367 | static void | | | |
| 1368 | process_other_stuff(recv) | | | |
| 1369 | register RECV_MODE *recv; | | | |
| 1370 | { | | | |
| 1371 | int err; | | | |
| 1372 | | | | |
| 1373 | /* | | | |
| 1374 | ** Inform task of packet arrival and let the ISR know | | | |
| 1375 | ** we made a system call so it uses UI_EXIT to leave ISR. | | | |
| 1376 | */ | | | |
| 1377 | if (copy_from_lance_ring(recv) == FAILURE) { | | | |
| 1378 | too_many_buffers = 1; | | | |
| 1379 | outputRoutine("WR"); /* No Buffers */ | | | |
| 1380 | return; /* non-fatal error */ | | | |
| 1381 | } | | | |
| 1382 | | | | |
| 1383 | system_call = 1; | | | |
| 1384 | if (extra_rcv_desc_count == 1) { | | | |
| 1385 | sc_spost (rcv_lance_pkt_semaphore, &err); | | | |
| 1386 | if (err) { | | | |
| 1387 | /* Turn the buffer back over to the LANCE. */ | | | |
| 1388 | recv->own_buffer = LANCE_BUFFER_OWNERSHIP; | | | |
| 1389 | | | | |
| 1390 | if (++current_receive_pointer >= maximum_receive_pointer) | | | |
| 1391 | current_receive_pointer = minimum_receive_pointer; | | | |
| 1392 | | | | |
| 1393 | outputRoutine("SP"); /* Semaphore Post */ | | | |
| 1394 | return; /* non-fatal error */ | | | |
| 1395 | } | | | |
| 1396 | | | | |
| 1397 | | | | |
| 1398 | | | | |
| 1399 | | | | |
| 1400 | static void | | | |
| 1401 | memory_copy(s1, s2, count) | | | |
| 1402 | register u_long *s1; | | | |
| 1403 | register u_long *s2; | | | |
| 1404 | register short count; | | | |
| 1405 | { | | | |
| 1406 | register short remain = count & 3; | | | |
| 1407 | | | | |
| 1408 | /* | | | |
| 1409 | ** divide by 4. | | | |
| 1410 | ** How many long moves we need to perform | | | |
| 1411 | */ | | | |
| 1412 | count >>= 2; | | | |
| 1413 | while(--count >= 0) { | | | |
| 1414 | *s1++ = *s2++; | | | |
| 1415 | } | | | |
| 1416 | /* | | | |
| 1417 | ** clean up, by moving remaining bytes | | | |
| 1418 | */ | | | |
| 1419 | while(--remain >= 0) { | | | |
| 1420 | *(u_char *)s1++ = *(u_char *)s2++; | | | |
| 1421 | } | | | |
| 1422 | | | | |
| 1423 | | | | |
| 1424 | | | | |
| 1425 | /* | | | |
| 1426 | * LANCE Interrupt Service Routine | | | |
| 1427 | */ | | | |
| 1428 | void | | | |
| 1429 | lance_isr() | | | |
| 1430 | { | | | |
| 1431 | u_short error; | | | |
| 1432 | u_short reset = 0; | | | |
| 1433 | u_short not_a_register_car0; | | | |
| 1434 | | | | |
| 1435 | register u_short car0; | | | |
| 1436 | register TRANS_MODE *trans_ptr; | | | |
| 1437 | | | | |
| 1438 | | | | |
| 1439 | /* | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lance.c | DATE 5/23/89 TIME 4:42:12 pm | PAGE # 13/29 |
|--|---|------------------------------|---------------------------------------|-----------------|
| SOURCE TEXT | | | | |
| LINE # | | | | |
| 1440 | /* Check the status of LANCE control and status register 0 | | | |
| 1441 | /* for possible errors and the source of the interrupt. | | | |
| 1442 | */ | | | |
| 1443 | if (lance_read(¬_a_register_car0) != SUCCESS) { | | | |
| 1444 | reset = 1; | | | |
| 1445 | outputRoutine("CSR"); | | | |
| 1446 | not_a_register_car0 = 0x0; | | | |
| 1447 | /* I know errors are gross, but we don't want to | | | |
| 1448 | /* check the stuff below if the CSR read failed. | | | |
| 1449 | */ | | | |
| 1450 | goto close_up_shop; | | | |
| 1451 | } | | | |
| 1452 | | | | |
| 1453 | | | | |
| 1454 | /* | | | |
| 1455 | /* Create a local register variable for car0. | | | |
| 1456 | */ | | | |
| 1457 | car0 = not_a_register_car0; | | | |
| 1458 | | | | |
| 1459 | /* | | | |
| 1460 | /* If we didn't get an interrupt, just harmlessly return. | | | |
| 1461 | /* In reality, something is seriously broken - oh well. | | | |
| 1462 | */ | | | |
| 1463 | if((car0 & INTERRUPT_SUMMARY) != INTERRUPT_SUMMARY) | | | |
| 1464 | return; | | | |
| 1465 | | | | |
| 1466 | /* | | | |
| 1467 | /* Check all the conditions that could have occurred. | | | |
| 1468 | /* Note that there are several fatal error conditions. | | | |
| 1469 | */ | | | |
| 1470 | if ((car0 & ERROR_SUMMARY) == ERROR_SUMMARY) { | | | |
| 1471 | /* Note that we don't reset here because if the error is | | | |
| 1472 | /* fatal the receiver and/or transmitter will be off and | | | |
| 1473 | /* checked below. | | | |
| 1474 | */ | | | |
| 1475 | if ((car0 & TRANSMIT_BABBLE_ERROR) == TRANSMIT_BABBLE_ERROR) | | | |
| 1476 | outputRoutine("BAB"); | | | |
| 1477 | if ((car0 & MISSED_PACKET_ERROR) == MISSED_PACKET_ERROR) | | | |
| 1478 | { | | | |
| 1479 | /* outputRoutine("MISS"); */ | | | |
| 1480 | } | | | |
| 1481 | | | | |
| 1482 | if ((car0 & MEMORY_ERROR) == MEMORY_ERROR) | | | |
| 1483 | { | | | |
| 1484 | /* outputRoutine("MEM"); */ | | | |
| 1485 | } | | | |
| 1486 | | | | |
| 1487 | /* COLLISION_ERROR | | | |
| 1488 | /* According to the LANCE technical manual, P4-15, collision error | | | |
| 1489 | /* should not be considered fatal. Comment out this test for | | | |
| 1490 | /* now but give it a try during system debug/test. | | | |
| 1491 | */ | | | |
| 1492 | if ((car0 & COLLISION_ERROR) == COLLISION_ERROR) | | | |
| 1493 | { | | | |
| 1494 | /* outputRoutine("COL"); */ | | | |
| 1495 | } | | | |
| 1496 | | | | |
| 1497 | | | | |
| 1498 | | | | |
| 1499 | if ((car0 & RECEIVE_ON) != RECEIVE_ON) { | | | |
| 1500 | reset = 1; | | | |
| 1501 | outputRoutine("RX"); | | | |
| 1502 | } | | | |
| 1503 | | | | |
| 1504 | if ((car0 & TRANSMIT_ON) != TRANSMIT_ON) { | | | |
| 1505 | reset = 1; | | | |
| 1506 | outputRoutine("TX"); | | | |
| 1507 | } | | | |
| 1508 | | | | |
| 1509 | if (((car0 & RECEIVE_INTERRUPT) == RECEIVE_INTERRUPT) | | | |
| 1510 | /* too many buffers at (extra_rcv_desc_count != MAX_USERS)) | | | |
| 1511 | { | | | |
| 1512 | lance_lsr_receive(&error); | | | |
| 1513 | } | | | |
| 1514 | | | | |
| 1515 | if ((car0 & TRANSMIT_INTERRUPT) == TRANSMIT_INTERRUPT) { | | | |
| 1516 | trans_ptr = lance_send_packet; | | | |
| 1517 | if (trans_ptr->max_packet >= maximum_transmit_pointer) { | | | |
| 1518 | lance_send_packet = minimum_transmit_pointer; | | | |
| 1519 | } | | | |
| 1520 | if(trans_ptr->own_buffer == HOST_BUFFER_OWNERSHIP) { | | | |
| 1521 | if(trans_ptr->buffer_error == 1) | | | |
| 1522 | outputRoutine("BUFF"); | | | |
| 1523 | if(trans_ptr->error_summary == 1) | | | |
| 1524 | { | | | |
| 1525 | if(trans_ptr->underflow_error == 1) | | | |
| 1526 | { | | | |
| 1527 | #ifdef BROKEN_HARDWARE | | | |
| 1528 | reinitialize_lance = 1; | | | |
| 1529 | /* (cpu_control_reg_struct *) CPU_CONTROL_REG->control_spare = 1; | | | |
| 1530 | #endif BROKEN_HARDWARE | | | |
| 1531 | outputRoutine("UFLO"); | | | |
| 1532 | } | | | |
| 1533 | if(trans_ptr->late_collision_error == 1) | | | |
| 1534 | { | | | |
| 1535 | /* outputRoutine("LCOL"); */ | | | |
| 1536 | } | | | |
| 1537 | | | | |
| 1538 | if(trans_ptr->lost_carrier_error == 1) | | | |
| 1539 | outputRoutine("LCAR"); | | | |
| 1540 | if(trans_ptr->retry_error == 1) | | | |
| 1541 | { | | | |
| 1542 | /* outputRoutine("RT"); */ | | | |
| 1543 | } | | | |
| 1544 | | | | |
| 1545 | | | | |
| 1546 | | | | |
| 1547 | | | | |
| 1548 | | | | |
| 1549 | close_up_shop: | | | |
| 1550 | if(reset == 1) { | | | |
| 1551 | (void)lance_access((char *) ¬_a_register_car0, LANCE_CSR_REG, sizeof_LANCE_CSR_REG, MEMORY_WRITE, &error); | | | |
| 1552 | reset_cpu(LANCE_ERROR); | | | |
| 1553 | } | | | |
| 1554 | | | | |
| 1555 | | | | |
| 1556 | /* | | | |
| 1557 | /* exchange buffers | | | |
| 1558 | /* between lance rcv ring & our spare ring | | | |
| 1559 | /* This avoids the situation of having to copy buffers in the ISR | | | |

| Copyright 1989 Logic Modeling Systems | SOURCE PROGRAM os/lance.c | DATE 5/23/89 TIME 4:42:12 pm | PAGE # 14/30 |
|--|---|---------------------------------|-----------------|
| LINE # | SOURCE TEXT | | |
| 1560 | /* | | |
| 1561 | copy from lance ring(rcv) | | |
| 1562 | register RECV_MSE "rcv; | | |
| 1563 | { | | |
| 1564 | register RECV_MSE "new_rcv = &extra_rcv_desc_buffers[in_extra_rcv_desc_ptr]; | | |
| 1565 | /* | | |
| 1566 | any buffers? | | |
| 1567 | /* | | |
| 1568 | if((available_extra_buffer < 0) (extra_rcv_desc_count == MAX_USERS)) | | |
| 1569 | return(FAILURE); | | |
| 1570 | /* | | |
| 1571 | copy the rcv descriptor | | |
| 1572 | /* | | |
| 1573 | new_rcv = "rcv; | | |
| 1574 | /* | | |
| 1575 | bump the "pointer", and the count | | |
| 1576 | /* | | |
| 1577 | in_extra_rcv_desc_ptr++; | | |
| 1578 | in_extra_rcv_desc_ptr = (MAX_USERS - 1); | | |
| 1579 | extra_rcv_desc_count++; | | |
| 1580 | /* | | |
| 1581 | give lance one of our spare buffers | | |
| 1582 | /* | | |
| 1583 | rcv->low_buffer_address = | | |
| 1584 | (u_short) p_la_rcv_buffers_extra[available_extra_buffer]; | | |
| 1585 | rcv->high_buffer_address = | | |
| 1586 | (unsigned) p_la_rcv_buffers_extra[available_extra_buffer - 1] >> 16; | | |
| 1587 | return(SUCCESS); | | |
| 1588 | } | | |
| 1589 | #ifdef ICMP_MESSAGES | | |
| 1590 | #define PR(number) { int i; for(i=0; i<number; ++i) { (void)printf("%02x", *ptr++); (void)printf("\n"); } | | |
| 1591 | static | | |
| 1592 | print_garbage_header(message) | | |
| 1593 | char *message; | | |
| 1594 | { | | |
| 1595 | register u_char *ptr; | | |
| 1596 | { | | |
| 1597 | ptr = (u_char *)message; | | |
| 1598 | (void)printf("received garbage\n"); | | |
| 1599 | (void)printf("Eset source: "); PR(6) | | |
| 1600 | (void)printf("Eset destination: "); PR(6) | | |
| 1601 | (void)printf("Eset type: "); PR(2) | | |
| 1602 | (void)printf("Eset V.IEL.Type: "); PR(2) | | |
| 1603 | (void)printf("Eset length: "); PR(2) | | |
| 1604 | (void)printf("Eset Id.Frag: "); PR(4) | | |
| 1605 | (void)printf("Eset TTL.Proc: "); PR(2) | | |
| 1606 | (void)printf("Eset Chksum: "); PR(2) | | |
| 1607 | (void)printf("Eset source: "); PR(4) | | |
| 1608 | (void)printf("Eset destination: "); PR(4) | | |
| 1609 | if (*((u_char *)message + 23) != 0x1) { | | |
| 1610 | (void)printf("ERROR: unknown protocol(0x%02x)\n", *(ptr+23)); | | |
| 1611 | } | | |
| 1612 | else { | | |
| 1613 | (void)printf("ICMP Type.Code: "); PR(2) | | |
| 1614 | (void)printf("ICMP Chksum: "); PR(2) | | |
| 1615 | (void)printf("ICMP unused: "); PR(4) | | |
| 1616 | (void)printf("Eset V.IEL.Type: "); PR(2) | | |
| 1617 | (void)printf("Eset length: "); PR(2) | | |
| 1618 | (void)printf("Eset Id.Frag: "); PR(4) | | |
| 1619 | (void)printf("Eset TTL.Proc: "); PR(2) | | |
| 1620 | (void)printf("Eset Chksum: "); PR(2) | | |
| 1621 | (void)printf("Eset source: "); PR(4) | | |
| 1622 | (void)printf("Eset destination: "); PR(4) | | |
| 1623 | } | | |
| 1624 | #endif ICMP_MESSAGES | | |
| 1625 | send_icmp_reply(rcv_buf) | | |
| 1626 | register char *rcv_buf; | | |
| 1627 | { | | |
| 1628 | extern ID_PROM_CPU id_prom; | | |
| 1629 | u_long inet_address, error; | | |
| 1630 | register PACKET_HEADER *rcv_buffer = (PACKET_HEADER *)rcv_buf; | | |
| 1631 | register u_long i, csum; | | |
| 1632 | register u_short j; | | |
| 1633 | register u_char *eset_address = (u_char *) id_prom.ethernet; | | |
| 1634 | register ICMP_HEADER *icmp_pkt = (ICMP_HEADER *) (rcv_buf + sizeof(ETHERNET_HEADER) + sizeof(IP_HEADER)); | | |
| 1635 | register u_short *icmp_checksum = (u_short *) (rcv_buf + sizeof(ETHERNET_HEADER) + sizeof(IP_HEADER)); | | |
| 1636 | /* | | |
| 1637 | xchg source destination ethernet addresses | | |
| 1638 | /* | | |
| 1639 | rcv_buffer->ethernet_hdr.destination[0]=rcv_buffer->ethernet_hdr.source[0]; | | |
| 1640 | rcv_buffer->ethernet_hdr.destination[1]=rcv_buffer->ethernet_hdr.source[1]; | | |
| 1641 | rcv_buffer->ethernet_hdr.destination[2]=rcv_buffer->ethernet_hdr.source[2]; | | |
| 1642 | rcv_buffer->ethernet_hdr.destination[3]=rcv_buffer->ethernet_hdr.source[3]; | | |
| 1643 | rcv_buffer->ethernet_hdr.destination[4]=rcv_buffer->ethernet_hdr.source[4]; | | |
| 1644 | rcv_buffer->ethernet_hdr.destination[5]=rcv_buffer->ethernet_hdr.source[5]; | | |
| 1645 | rcv_buffer->ethernet_hdr.source[0]=eset_address[0]; | | |
| 1646 | rcv_buffer->ethernet_hdr.source[1]=eset_address[1]; | | |
| 1647 | rcv_buffer->ethernet_hdr.source[2]=eset_address[2]; | | |
| 1648 | rcv_buffer->ethernet_hdr.source[3]=eset_address[3]; | | |
| 1649 | rcv_buffer->ethernet_hdr.source[4]=eset_address[4]; | | |
| 1650 | rcv_buffer->ethernet_hdr.source[5]=eset_address[5]; | | |
| 1651 | j = rcv_buffer->ip_hdr.total_length - sizeof(IP_HEADER); | | |
| 1652 | /* | | |
| 1653 | xchg source destination internet addresses. | | |
| 1654 | /* | | |
| 1655 | inet_address = rcv_buffer->ip_hdr.source_address; | | |
| 1656 | rcv_buffer->ip_hdr.source_address=rcv_buffer->ip_hdr.destination_address; | | |
| 1657 | rcv_buffer->ip_hdr.destination_address = inet_address; | | |
| 1658 | /* | | |
| 1659 | set reply type | | |
| 1660 | /* | | |
| 1661 | icmp_pkt->type = 0; | | |
| 1662 | /* | | |
| 1663 | calculate ICMP checksum | | |
| 1664 | /* | | |
| 1665 | csum = 0; | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lance.c | DATE 5/23/89 | PAGE # 15/31 |
|--|--|---|-----------------|-----------------|
| LINE # | | SOURCE TEXT | | |
| 1680 | | icmp_pkt->checksum = 0; | | |
| 1681 | | for(i = 0; i < 3; i++) | | |
| 1682 | | checksum += icmp_checksum++; | | |
| 1683 | | checksum >> 16; /* add in the carry */ | | |
| 1684 | | icmp_pkt->checksum = ((u_short) checksum); /* the one's complement of the checksum */ | | |
| 1685 | | lance_transmit(recv_buf, recv_buffer->ip_hdr.total_length | | |
| 1686 | | + sizeof(ETHERNET_HEADER), &error); | | |
| 1687 | |) | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lm_ee_access.c | DATE 5/23/89 | PAGE # 1/32 |
|--|--|-------------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCCS ID: lm_ee_access.c rev 3.1, 4/24/89 at 07:46:54 */ | | | |
| 2 | #include "common.h" | | | |
| 3 | #include "mod_def.h" | | | |
| 4 | #include "cpu.h" | | | |
| 5 | #include "magic.h" | | | |
| 6 | #include "pel.h" | | | |
| 7 | #include "lm_rd_wr.h" | | | |
| 8 | #include "eeprom.h" | | | |
| 9 | | | | |
| 10 | | | | |
| 11 | u_short access_eeprom(); | | | |
| 12 | u_short write_eeprom(), read_eeprom(); | | | |
| 13 | void disable_eeprom(), enable_eeprom(), erase_all_eeprom(); | | | |
| 14 | u_char diag_dab; | | | |
| 15 | u_short | | | |
| 16 | lm_eeprom_access(data_struct, field, ee_num, sizeof_field, option, error) | | | |
| 17 | char *data_struct; | | | |
| 18 | unsigned long field; | | | |
| 19 | u_long ee_num; | | | |
| 20 | unsigned long sizeof_field, option; | | | |
| 21 | unsigned long *error; | | | |
| 22 | { | | | |
| 23 | register PEL *pel_access_reg; | | | |
| 24 | u_short status; | | | |
| 25 | register u_char ee_number = ee_num; | | | |
| 26 | register unsigned char eeprom = (unsigned char) field; | | | |
| 27 | register unsigned short checksum = 0; | | | |
| 28 | register unsigned short temp, i = 0, *ptr = (u_short *) data_struct; | | | |
| 29 | | | | |
| 30 | u_char old_eeprom_in_bit; | | | |
| 31 | u_char old_initialize_bit; | | | |
| 32 | | | | |
| 33 | *error = NO_NVRAM_ERROR; | | | |
| 34 | | | | |
| 35 | pel_access_reg = (PEL *) | | | |
| 36 | (pel_addr(((ee_number & 0x15) >> 3), (ee_number & 7))); | | | |
| 37 | | | | |
| 38 | /* | | | |
| 39 | ** check for diag dab by toggling eeprom in | | | |
| 40 | ** and present should follow. | | | |
| 41 | */ | | | |
| 42 | diag_dab = 0; | | | |
| 43 | old_eeprom_in_bit = pel_access_reg->car.bit.eeprom_in; | | | |
| 44 | pel_access_reg->car.bit.eeprom_in = 1; | | | |
| 45 | if(pel_access_reg->car.bit.present == 0) | | | |
| 46 | { | | | |
| 47 | pel_access_reg->car.bit.eeprom_in = 0; | | | |
| 48 | if(pel_access_reg->car.bit.present == 1) | | | |
| 49 | { | | | |
| 50 | diag_dab = 1; | | | |
| 51 | } | | | |
| 52 | } | | | |
| 53 | pel_access_reg->car.bit.eeprom_in = old_eeprom_in_bit; | | | |
| 54 | old_initialize_bit = pel_access_reg->car.bit.initialize; | | | |
| 55 | pel_access_reg->car.bit.initialize = 0; | | | |
| 56 | pel_access_reg->car.bit.initialize = 1; | | | |
| 57 | /* | | | |
| 58 | ** if reading or writing we have valid eeprom | | | |
| 59 | ** and eeprom access is within bounds | | | |
| 60 | */ | | | |
| 61 | if(option == MEMORY_READ option == MEMORY_WRITE) | | | |
| 62 | { | | | |
| 63 | if(field + sizeof_field > (CPU_EEPROM_SIZE)) | | | |
| 64 | { | | | |
| 65 | *error = NO_NVRAM; | | | |
| 66 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 67 | return(FAILURE); | | | |
| 68 | } | | | |
| 69 | } | | | |
| 70 | /* | | | |
| 71 | ** perform read | | | |
| 72 | */ | | | |
| 73 | if(option == MEMORY_READ) | | | |
| 74 | { | | | |
| 75 | /* | | | |
| 76 | ** take care of odd reads | | | |
| 77 | */ | | | |
| 78 | if(eeprom & 1) | | | |
| 79 | { | | | |
| 80 | *(u_char *) ptr = (u_char) (read_eeprom(ee_number, eeprom >> 1, &status) >> 8); | | | |
| 81 | if(status == FAILURE) { | | | |
| 82 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 83 | return(FAILURE); | | | |
| 84 | } | | | |
| 85 | ++eeprom; | | | |
| 86 | (u_char *) ++ptr; | | | |
| 87 | --sizeof_field; | | | |
| 88 | } | | | |
| 89 | eeprom >>= 1; | | | |
| 90 | /* | | | |
| 91 | ** Anything else to read | | | |
| 92 | */ | | | |
| 93 | if(sizeof_field) | | | |
| 94 | { | | | |
| 95 | /* | | | |
| 96 | ** skip odd read at the end | | | |
| 97 | */ | | | |
| 98 | temp = sizeof_field & 0xffff; | | | |
| 99 | if(temp) | | | |
| 100 | { | | | |
| 101 | for(i = 0; i < temp; i+=2) | | | |
| 102 | { | | | |
| 103 | *ptr++ = read_eeprom(ee_number, eeprom, &status); | | | |
| 104 | if(status == FAILURE) { | | | |
| 105 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 106 | return(FAILURE); | | | |
| 107 | } | | | |
| 108 | eeprom ++; | | | |
| 109 | } | | | |
| 110 | } | | | |
| 111 | /* | | | |
| 112 | ** do odd read at the end | | | |
| 113 | */ | | | |
| 114 | if(sizeof_field & 1) | | | |
| 115 | { | | | |
| 116 | *(u_char *) ptr = (u_char) (read_eeprom(ee_number, eeprom, &status)); | | | |
| 117 | if(status == FAILURE) { | | | |
| 118 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 119 | return(FAILURE); | | | |
| 120 | } | | | |
| 121 | } | | | |

| | | | |
|---|---|--|--|
| <div> <div>Copyright 1989</div> <div>Logic Modeling Systems</div> </div> | <div> <div>SOURCE PROGRAM</div> <div>os/lm_ee_access.c</div> </div> | <div> <div>DATE</div> <div>5/23/89</div> </div> <div> <div>TIME</div> <div>4:42:13 pm</div> </div> | <div> <div>PAGE #</div> <div>2/33</div> </div> |
| | | | |
| <div>LINE #</div> <div>121</div> <div>122</div> <div>123</div> <div>124</div> <div>125</div> <div>126</div> <div>127</div> <div>128</div> <div>129</div> <div>130</div> <div>131</div> <div>132</div> <div>133</div> <div>134</div> <div>135</div> <div>136</div> <div>137</div> <div>138</div> <div>139</div> <div>140</div> <div>141</div> <div>142</div> <div>143</div> <div>144</div> <div>145</div> <div>146</div> <div>147</div> <div>148</div> <div>149</div> <div>150</div> <div>151</div> <div>152</div> <div>153</div> <div>154</div> <div>155</div> <div>156</div> <div>157</div> <div>158</div> <div>159</div> <div>160</div> <div>161</div> <div>162</div> <div>163</div> <div>164</div> <div>165</div> <div>166</div> <div>167</div> <div>168</div> <div>169</div> <div>170</div> <div>171</div> <div>172</div> <div>173</div> <div>174</div> <div>175</div> <div>176</div> <div>177</div> <div>178</div> <div>179</div> <div>180</div> <div>181</div> <div>182</div> <div>183</div> <div>184</div> <div>185</div> <div>186</div> <div>187</div> <div>188</div> <div>189</div> <div>190</div> <div>191</div> <div>192</div> <div>193</div> <div>194</div> <div>195</div> <div>196</div> <div>197</div> <div>198</div> <div>199</div> <div>200</div> <div>201</div> <div>202</div> <div>203</div> <div>204</div> <div>205</div> <div>206</div> <div>207</div> <div>208</div> <div>209</div> <div>210</div> <div>211</div> <div>212</div> <div>213</div> <div>214</div> <div>215</div> <div>216</div> <div>217</div> <div>218</div> <div>219</div> <div>220</div> <div>221</div> <div>222</div> <div>223</div> <div>224</div> <div>225</div> <div>226</div> <div>227</div> <div>228</div> <div>229</div> <div>230</div> <div>231</div> <div>232</div> <div>233</div> <div>234</div> <div>235</div> <div>236</div> <div>237</div> <div>238</div> <div>239</div> <div>240</div> <div>241</div> <div>242</div> <div>243</div> <div>244</div> <div>245</div> <div>246</div> <div>247</div> <div>248</div> <div>249</div> <div>250</div> <div>251</div> <div>252</div> <div>253</div> <div>254</div> <div>255</div> <div>256</div> <div>257</div> <div>258</div> <div>259</div> <div>260</div> <div>261</div> <div>262</div> <div>263</div> <div>264</div> <div>265</div> <div>266</div> <div>267</div> <div>268</div> <div>269</div> <div>270</div> <div>271</div> <div>272</div> <div>273</div> <div>274</div> <div>275</div> <div>276</div> <div>277</div> <div>278</div> <div>279</div> <div>280</div> <div>281</div> <div>282</div> <div>283</div> <div>284</div> <div>285</div> <div>286</div> <div>287</div> <div>288</div> <div>289</div> <div>290</div> <div>291</div> <div>292</div> <div>293</div> <div>294</div> <div>295</div> <div>296</div> <div>297</div> <div>298</div> <div>299</div> <div>300</div> <div>301</div> <div>302</div> <div>303</div> <div>304</div> <div>305</div> <div>306</div> <div>307</div> <div>308</div> <div>309</div> <div>310</div> <div>311</div> <div>312</div> <div>313</div> <div>314</div> <div>315</div> <div>316</div> <div>317</div> <div>318</div> <div>319</div> <div>320</div> <div>321</div> <div>322</div> <div>323</div> <div>324</div> <div>325</div> <div>326</div> <div>327</div> <div>328</div> <div>329</div> <div>330</div> <div>331</div> <div>332</div> <div>333</div> <div>334</div> <div>335</div> <div>336</div> <div>337</div> <div>338</div> <div>339</div> <div>340</div> <div>341</div> <div>342</div> <div>343</div> <div>344</div> <div>345</div> <div>346</div> <div>347</div> <div>348</div> <div>349</div> <div>350</div> <div>351</div> <div>352</div> <div>353</div> <div>354</div> <div>355</div> <div>356</div> <div>357</div> <div>358</div> <div>359</div> <div>360</div> <div>361</div> <div>362</div> <div>363</div> <div>364</div> <div>365</div> <div>366</div> <div>367</div> <div>368</div> <div>369</div> <div>370</div> <div>371</div> <div>372</div> <div>373</div> <div>374</div> <div>375</div> <div>376</div> <div>377</div> <div>378</div> <div>379</div> <div>380</div> <div>381</div> <div>382</div> <div>383</div> <div>384</div> <div>385</div> <div>386</div> <div>387</div> <div>388</div> <div>389</div> <div>390</div> <div>391</div> <div>392</div> <div>393</div> <div>394</div> <div>395</div> <div>396</div> <div>397</div> <div>398</div> <div>399</div> <div>400</div> <div>401</div> <div>402</div> <div>403</div> <div>404</div> <div>405</div> <div>406</div> <div>407</div> <div>408</div> <div>409</div> <div>410</div> <div>411</div> <div>412</div> <div>413</div> <div>414</div> <div>415</div> <div>416</div> <div>417</div> <div>418</div> <div>419</div> <div>420</div> <div>421</div> <div>422</div> <div>423</div> <div>424</div> <div>425</div> <div>426</div> <div>427</div> <div>428</div> <div>429</div> <div>430</div> <div>431</div> <div>432</div> <div>433</div> <div>434</div> <div>435</div> <div>436</div> <div>437</div> <div>438</div> <div>439</div> <div>440</div> <div>441</div> <div>442</div> <div>443</div> <div>444</div> <div>445</div> <div>446</div> <div>447</div> <div>448</div> <div>449</div> <div>450</div> <div>451</div> <div>452</div> <div>453</div> <div>454</div> <div>455</div> <div>456</div> <div>457</div> <div>458</div> <div>459</div> <div>460</div> <div>461</div> <div>462</div> <div>463</div> <div>464</div> <div>465</div> <div>466</div> <div>467</div> <div>468</div> <div>469</div> <div>470</div> <div>471</div> <div>472</div> <div>473</div> <div>474</div> <div>475</div> <div>476</div> <div>477</div> <div>478</div> <div>479</div> <div>480</div> <div>481</div> <div>482</div> <div>483</div> <div>484</div> <div>485</div> <div>486</div> <div>487</div> <div>488</div> <div>489</div> <div>490</div> <div>491</div> <div>492</div> <div>493</div> <div>494</div> <div>495</div> <div>496</div> <div>497</div> <div>498</div> <div>499</div> <div>500</div> <div>501</div> <div>502</div> <div>503</div> <div>504</div> <div>505</div> <div>506</div> <div>507</div> <div>508</div> <div>509</div> <div>510</div> <div>511</div> <div>512</div> <div>513</div> <div>514</div> <div>515</div> <div>516</div> <div>517</div> <div>518</div> <div>519</div> <div>520</div> <div>521</div> <div>522</div> <div>523</div> <div>524</div> <div>525</div> <div>526</div> <div>527</div> <div>528</div> <div>529</div> <div>530</div> <div>531</div> <div>532</div> <div>533</div> <div>534</div> | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lm_ee_access.c | DATE * 5/23/89 TIME 4:42:13 pm | PAGE # 3/34 |
|--|--|-------------------------------------|-----------------------------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 239 | /*read*/ write_eeprom(ee_number, 1, (u_short) 0); | | | |
| 240 | /* set up signature | | | |
| 241 | */ | | | |
| 242 | if(write_eeprom(ee_number, EEPROM_SIGNATURE_0, (u_short) SIG_WORD_0) == FAILURE) | | | |
| 243 | { | | | |
| 244 | /*error = WRITE FAILURE; | | | |
| 245 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 246 | return(FAILURE); | | | |
| 247 | } | | | |
| 248 | /* | | | |
| 249 | /* set up signature | | | |
| 250 | */ | | | |
| 251 | if(write_eeprom(ee_number, EEPROM_SIGNATURE_1, (u_short) SIG_WORD_1) == FAILURE) | | | |
| 252 | { | | | |
| 253 | /*error = WRITE FAILURE; | | | |
| 254 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 255 | return(FAILURE); | | | |
| 256 | } | | | |
| 257 | /* | | | |
| 258 | /* This assumes checksum and write count are on a even boundary | | | |
| 259 | */ | | | |
| 260 | if(write_eeprom(ee_number, EEPROM_WRITE_COUNT, (u_short) read_eeprom(ee_number, EEPROM_WRITE_COUNT)+1, &status) == FAILURE) | | | |
| 261 | { | | | |
| 262 | /*error = WRITE FAILURE; | | | |
| 263 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 264 | return(FAILURE); | | | |
| 265 | } | | | |
| 266 | /* | | | |
| 267 | /* calculate checksum, do not use checksum in the calculation | | | |
| 268 | /* 2's complement of the sum of the rest of the fields | | | |
| 269 | */ | | | |
| 270 | for(i = 0; i != 63; i++) | | | |
| 271 | { | | | |
| 272 | checksum += read_eeprom(ee_number, i, &status); | | | |
| 273 | if(status == FAILURE) { | | | |
| 274 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 275 | return(FAILURE); | | | |
| 276 | } | | | |
| 277 | } | | | |
| 278 | if(write_eeprom(ee_number, EEPROM_CHECKSUM, (u_short) ((~checksum) + 1)) == FAILURE) | | | |
| 279 | { | | | |
| 280 | /*error = WRITE FAILURE; | | | |
| 281 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 282 | return(FAILURE); | | | |
| 283 | } | | | |
| 284 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 285 | return(SUCCESS); | | | |
| 286 | } | | | |
| 287 | /* | | | |
| 288 | /* validate NVRAM | | | |
| 289 | */ | | | |
| 290 | if(option == MEMORY_VALIDATE) | | | |
| 291 | { | | | |
| 292 | if(read_eeprom(ee_number, EEPROM_SIGNATURE_0, &status) != SIG_WORD_0) | | | |
| 293 | { | | | |
| 294 | /*error = INVALID_NVRAM; | | | |
| 295 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 296 | return(FAILURE); | | | |
| 297 | } | | | |
| 298 | if(status == FAILURE) { | | | |
| 299 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 300 | return(FAILURE); | | | |
| 301 | } | | | |
| 302 | if(read_eeprom(ee_number, EEPROM_SIGNATURE_1, &status) != SIG_WORD_1) | | | |
| 303 | { | | | |
| 304 | /*error = INVALID_NVRAM; | | | |
| 305 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 306 | return(FAILURE); | | | |
| 307 | } | | | |
| 308 | if(status == FAILURE) { | | | |
| 309 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 310 | return(FAILURE); | | | |
| 311 | } | | | |
| 312 | /* | | | |
| 313 | /* calculate checksum, do not use checksum in the calculation | | | |
| 314 | /* 2's complement of the sum of the rest of the fields | | | |
| 315 | */ | | | |
| 316 | for(i = 0; i != 64; i++) | | | |
| 317 | { | | | |
| 318 | /*ptr = read_eeprom(ee_number, i, &status); | | | |
| 319 | if(status == FAILURE) { | | | |
| 320 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 321 | return(FAILURE); | | | |
| 322 | } | | | |
| 323 | checksum += *ptr++; | | | |
| 324 | } | | | |
| 325 | if(checksum) | | | |
| 326 | { | | | |
| 327 | /*error = INVALID_NVRAM; | | | |
| 328 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 329 | return(FAILURE); | | | |
| 330 | } | | | |
| 331 | pel_access_reg->car.bit.initialize = old_initialize_bit; | | | |
| 332 | return(SUCCESS); | | | |
| 333 | } | | | |
| 334 | /* | | | |
| 335 | /* validate NVRAM | | | |
| 336 | */ | | | |
| 337 | return(FAILURE); | | | |
| 338 | } | | | |
| 339 | /* | | | |
| 340 | /* read_eeprom(ee_number, dab) | | | |
| 341 | /* | | | |
| 342 | /* long ee_number. | | | |
| 343 | DAB_EEPROM *dab; | | | |
| 344 | { | | | |
| 345 | /*long error. | | | |
| 346 | if(!bzero((char *)dab, sizeof(DAB_EEPROM))) | | | |
| 347 | { | | | |
| 348 | /* Don't initialize the EEPROM on read errors. | | | |
| 349 | lm_eeprom_access((char *)dab, (u_long) 0, ee_number, (u_long) sizeof(DAB_EEPROM) , MEMORY_INIT, &error); | | | |
| 350 | return(FAILURE); | | | |
| 351 | } | | | |
| 352 | return(SUCCESS); | | | |
| 353 | } | | | |
| 354 | /* | | | |
| 355 | /* validate NVRAM | | | |
| 356 | */ | | | |
| 357 | char trash = 'L'; | | | |
| 358 | u_short | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/lm_ee_access.c

DATE 5/23/89 PAGE #
TIME 4:42:13 pm 4/35

```

LINE # SOURCE TEXT
359 lm_write_eeeprom( ee_number, dab )
360 register u_long ee_number;
361 register DAB_EEPROM *dab;
362 {
363     u_long error;
364     u_short status;
365     DAB_EEPROM dab2;
366
367     if( lm_eeprom_access( (char *)&dab2, (u_long) 0, ee_number,
368         (u_long)sizeof(DAB_EEPROM), MEMORY_VALIDATE, &error)--FAILURE)
369     {
370         lm_eeprom_access( (char *)&dab2, (u_long) 0, ee_number,
371             (u_long )sizeof(DAB_EEPROM), MEMORY_INIT, &error);
372     }
373
374     dab->signature[ 0 ] = 'M';
375     dab->signature[ 1 ] = 'M';
376     dab->signature[ 2 ] = 'S';
377     dab->signature[ 3 ] = 'I';
378
379     dab->write_count = read_eeeprom( (u_char)ee_number,
380         EEPROM_WRITE_COUNT, &status );
381
382     return( lm_eeprom_access( (char *)&dab, (u_long) 0, ee_number,
383         (u_long)sizeof(DAB_EEPROM), MEMORY_WRITE, &error));
384 }
385
386 u_short
387 lm_write_eeeprom_ins_count( ee_number, count)
388 register u_long ee_number;
389 u_short count;
390 {
391     u_long error;
392     return( lm_eeprom_access((char *)&count, (u_long) (E2_INS_COUNT ), ee_number, (u_long )sizeof(short) , MEMORY_WRITE, &error));
393 }
394

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lm_rd_wr.c | DATE 5/23/89 | PAGE # 1/36 |
|--|---|---------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCCS ID: lm_rd_wr.c rev 3.1, 4/24/89 at 07:46:57 */ | | | |
| 2 | /* | | | |
| 3 | ** process_lm_read() | | | |
| 4 | ** read modeler memory | | | |
| 5 | */ | | | |
| 6 | | | | |
| 7 | #include "common.h" | | | |
| 8 | #include "cpu.h" | | | |
| 9 | #include "lm_rd_wr.h" | | | |
| 10 | #include "mod_err.h" | | | |
| 11 | #include "sparam.h" | | | |
| 12 | #include "id.h" | | | |
| 13 | #include "eprom.h" | | | |
| 14 | #include "uart.h" | | | |
| 15 | | | | |
| 16 | | | | |
| 17 | extern u_short lm_cpumem_access(); | | | |
| 18 | | | | |
| 19 | extern u_short lm_sparam_access(); | | | |
| 20 | extern u_short lm_eprom_access(); | | | |
| 21 | extern u_short lm_idprom_access(); | | | |
| 22 | extern u_short lm_modeler_access(); | | | |
| 23 | extern u_short lm_cpureg_access(); | | | |
| 24 | extern u_short lm_duarta_access(); | | | |
| 25 | extern u_short lm_duartb_access(); | | | |
| 26 | | | | |
| 27 | | | | |
| 28 | | | | |
| 29 | static char init = 0; | | | |
| 30 | static u_short (*rd_wr_routines[LM_MAX_MEMORY]()); | | | |
| 31 | static void | | | |
| 32 | init_lm_rd_wr() | | | |
| 33 | { | | | |
| 34 | rd_wr_routines[LM_CPURAM_MEMORY] = lm_cpumem_access; | | | |
| 35 | rd_wr_routines[LM_NVRAM_MEMORY] = lm_sparam_access; | | | |
| 36 | rd_wr_routines[LM_EPROM_MEMORY] = lm_eprom_access; | | | |
| 37 | rd_wr_routines[LM_IDPROM_MEMORY] = lm_idprom_access; | | | |
| 38 | rd_wr_routines[LM_MODELER_MEMORY] = lm_modeler_access; | | | |
| 39 | rd_wr_routines[LM_CPUREG_MEMORY] = lm_cpureg_access; | | | |
| 40 | rd_wr_routines[LM_DUARTA_MEMORY] = lm_duarta_access; | | | |
| 41 | rd_wr_routines[LM_DUARTB_MEMORY] = lm_duartb_access; | | | |
| 42 | init = 1; | | | |
| 43 | } | | | |
| 44 | | | | |
| 45 | /* | | | |
| 46 | ** read from modeler memory space | | | |
| 47 | */ | | | |
| 48 | u_short | | | |
| 49 | proc_lm_rd(memory_type, ee_number, offset, number_of_bytes, buffer, status) | | | |
| 50 | { | | | |
| 51 | register u_long memory_type; | | | |
| 52 | register u_long ee_number; | | | |
| 53 | register u_long offset; | | | |
| 54 | register u_long number_of_bytes; | | | |
| 55 | register char *buffer; | | | |
| 56 | register u_long *status; | | | |
| 57 | { | | | |
| 58 | /* | | | |
| 59 | ** Make sure function array is set up | | | |
| 60 | */ | | | |
| 61 | if(init == 0) | | | |
| 62 | init_lm_rd_wr(); | | | |
| 63 | | | | |
| 64 | /* | | | |
| 65 | ** do error checking | | | |
| 66 | */ | | | |
| 67 | if(memory_type > LM_MAX_MEMORY) | | | |
| 68 | { | | | |
| 69 | *status = INVALID_PARAMETER; | | | |
| 70 | return(FAILURE); | | | |
| 71 | } | | | |
| 72 | | | | |
| 73 | if(number_of_bytes > MAX_RD_WR_BUFFER_SIZE) | | | |
| 74 | { | | | |
| 75 | *status = INVALID_PARAMETER; | | | |
| 76 | return(FAILURE); | | | |
| 77 | } | | | |
| 78 | | | | |
| 79 | /* | | | |
| 80 | ** make the call | | | |
| 81 | */ | | | |
| 82 | if(memory_type == LM_EEPROM_MEMORY) | | | |
| 83 | return((*rd_wr_routines[LM_EEPROM_MEMORY])(buffer, offset, ee_number, number_of_bytes, MEMORY_READ, status)); | | | |
| 84 | else | | | |
| 85 | return((*rd_wr_routines[memory_type])(buffer, offset, number_of_bytes, MEMORY_READ, status)); | | | |
| 86 | | | | |
| 87 | } | | | |
| 88 | /* | | | |
| 89 | ** write to modeler memory space | | | |
| 90 | */ | | | |
| 91 | u_short | | | |
| 92 | proc_lm_wr(memory_type, ee_number, offset, number_of_bytes, buffer, status) | | | |
| 93 | { | | | |
| 94 | register u_long memory_type; | | | |
| 95 | register u_long ee_number; | | | |
| 96 | register u_long offset; | | | |
| 97 | register u_long number_of_bytes; | | | |
| 98 | register char *buffer; | | | |
| 99 | register u_long *status; | | | |
| 100 | { | | | |
| 101 | /* | | | |
| 102 | ** Make sure function array is set up | | | |
| 103 | */ | | | |
| 104 | if(init == 0) | | | |
| 105 | init_lm_rd_wr(); | | | |
| 106 | | | | |
| 107 | /* | | | |
| 108 | ** do error checking | | | |
| 109 | */ | | | |
| 110 | if(memory_type > LM_MAX_MEMORY) | | | |
| 111 | { | | | |
| 112 | *status = INVALID_PARAMETER; | | | |
| 113 | return(FAILURE); | | | |
| 114 | } | | | |
| 115 | | | | |
| 116 | if(number_of_bytes > MAX_RD_WR_BUFFER_SIZE) | | | |
| 117 | { | | | |
| 118 | *status = INVALID_PARAMETER; | | | |
| 119 | return(FAILURE); | | | |
| 120 | } | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lm_rd_wr.c | DATE 5/23/89 TIME 4:42:14 pm | PAGE # 2/37 |
|--|--|---------------------------------|---------------------------------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 121 | /* make the call | | | |
| 122 | /* | | | |
| 123 | if(memory_type == _2PROM_MEMORY) | | | |
| 124 | return((*rd_wr_routines[LM_2PROM_MEMORY])(buffer, offset, ee_number, number_of_bytes, MEMORY_WRITE, status)); | | | |
| 125 | /* | | | |
| 126 | else | | | |
| 127 | return((*rd_wr_routines[memory_type])(buffer, offset, number_of_bytes, MEMORY_WRITE, status)); | | | |
| 128 | /* | | | |
| 129 | /* | | | |
| 130 | /* access CPU RAM | | | |
| 131 | /* | | | |
| 132 | u_short lm_cpurn_access(buffer, offset, number_of_bytes, option, status) | | | |
| 133 | register u_long option; | | | |
| 134 | register u_long offset; | | | |
| 135 | register u_long number_of_bytes; | | | |
| 136 | register char *buffer; | | | |
| 137 | register u_long *status; | | | |
| 138 | { | | | |
| 139 | register char *src_ptr, *dst_ptr; | | | |
| 140 | { | | | |
| 141 | if((offset + number_of_bytes) >= CPU_RAM_SIZE) | | | |
| 142 | { | | | |
| 143 | *status = NO_MEMORY; | | | |
| 144 | return(FAILURE); | | | |
| 145 | /* | | | |
| 146 | /* set up pointers | | | |
| 147 | /* | | | |
| 148 | if(option == MEMORY_READ) | | | |
| 149 | { | | | |
| 150 | src_ptr = (char *) (CPU_RAM + offset); | | | |
| 151 | dst_ptr = buffer; | | | |
| 152 | /* | | | |
| 153 | if(option == MEMORY_WRITE) | | | |
| 154 | { | | | |
| 155 | dst_ptr = (char *) (CPU_RAM + offset); | | | |
| 156 | src_ptr = buffer; | | | |
| 157 | /* | | | |
| 158 | /* Do the copy | | | |
| 159 | /* | | | |
| 160 | while(number_of_bytes--) | | | |
| 161 | *dst_ptr++ = *src_ptr++; | | | |
| 162 | /* | | | |
| 163 | return(SUCCESS); | | | |
| 164 | /* | | | |
| 165 | /* | | | |
| 166 | /* | | | |
| 167 | /* | | | |
| 168 | /* | | | |
| 169 | /* | | | |
| 170 | /* | | | |
| 171 | /* access CPU eeprom | | | |
| 172 | /* | | | |
| 173 | u_short lm_eprom_access(buffer, offset, number_of_bytes, option, status) | | | |
| 174 | register u_long option; | | | |
| 175 | register u_long offset; | | | |
| 176 | register u_long number_of_bytes; | | | |
| 177 | register char *buffer; | | | |
| 178 | register u_long *status; | | | |
| 179 | { | | | |
| 180 | register char *src_ptr, *dst_ptr; | | | |
| 181 | register eeprom_struct *eprom; | | | |
| 182 | { | | | |
| 183 | if(option == MEMORY_WRITE) | | | |
| 184 | { | | | |
| 185 | *status = INVALID_PARAMETER; | | | |
| 186 | return(FAILURE); | | | |
| 187 | /* | | | |
| 188 | /* | | | |
| 189 | eprom = (eeprom_struct *) CPU_EPROM; | | | |
| 190 | if((offset + number_of_bytes) >= (CPU_EPROM + eeprom->eeprom_size)) | | | |
| 191 | { | | | |
| 192 | *status = NO_MEMORY; | | | |
| 193 | return(FAILURE); | | | |
| 194 | /* | | | |
| 195 | /* | | | |
| 196 | /* set up pointers | | | |
| 197 | /* | | | |
| 198 | if(option == MEMORY_READ) | | | |
| 199 | { | | | |
| 200 | src_ptr = (char *) (CPU_EPROM + offset); | | | |
| 201 | dst_ptr = buffer; | | | |
| 202 | /* | | | |
| 203 | /* | | | |
| 204 | /* | | | |
| 205 | /* Do the copy | | | |
| 206 | /* | | | |
| 207 | while(number_of_bytes--) | | | |
| 208 | *dst_ptr++ = *src_ptr++; | | | |
| 209 | /* | | | |
| 210 | return(SUCCESS); | | | |
| 211 | /* | | | |
| 212 | /* | | | |
| 213 | /* access CPU idprom | | | |
| 214 | /* | | | |
| 215 | u_short lm_idprom_access(buffer, offset, number_of_bytes, option, status) | | | |
| 216 | register u_long option; | | | |
| 217 | register u_long offset; | | | |
| 218 | register u_long number_of_bytes; | | | |
| 219 | register char *buffer; | | | |
| 220 | register u_long *status; | | | |
| 221 | { | | | |
| 222 | register char *src_ptr, *dst_ptr; | | | |
| 223 | extern id_prom_cpu id_prom; | | | |
| 224 | { | | | |
| 225 | if(option == MEMORY_WRITE) | | | |
| 226 | { | | | |
| 227 | *status = INVALID_PARAMETER; | | | |
| 228 | return(FAILURE); | | | |
| 229 | /* | | | |
| 230 | /* | | | |
| 231 | /* | | | |
| 232 | if((offset + number_of_bytes) >= CPU_ID_PROM_SIZE) | | | |
| 233 | { | | | |
| 234 | *status = NO_MEMORY; | | | |
| 235 | return(FAILURE); | | | |
| 236 | /* | | | |
| 237 | /* | | | |
| 238 | /* | | | |
| 239 | /* set up pointers | | | |
| 240 | /* | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lm_rd_wr.c | DATE 5/23/89 | PAGE # 3/38 |
|--|---|---------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 241 | if(option == MEMORY_READ) | | | |
| 242 | { | | | |
| 243 | arc_ptr = ((char *) &id_prom) + offset; | | | |
| 244 | dst_ptr = buffer; | | | |
| 245 | } | | | |
| 246 | /* | | | |
| 247 | Do the copy | | | |
| 248 | */ | | | |
| 249 | while(number_of_bytes--) | | | |
| 250 | dst_ptr++ = *arc_ptr++; | | | |
| 251 | return(SUCCESS); | | | |
| 252 | } | | | |
| 253 | /* | | | |
| 254 | access modeler | | | |
| 255 | */ | | | |
| 256 | u_short lm_modeler_access(buffer, offset, number_of_bytes, option, status) | | | |
| 257 | register u_long option; | | | |
| 258 | register u_long offset; | | | |
| 259 | register u_long number_of_bytes; | | | |
| 260 | register char *buffer; | | | |
| 261 | register u_long *status; | | | |
| 262 | { | | | |
| 263 | register char *arc_ptr, *dst_ptr; | | | |
| 264 | /* | | | |
| 265 | set up pointers | | | |
| 266 | */ | | | |
| 267 | if(option == MEMORY_READ) | | | |
| 268 | { | | | |
| 269 | arc_ptr = (char *) (CPU_RAM + offset); | | | |
| 270 | dst_ptr = buffer; | | | |
| 271 | } | | | |
| 272 | if(option == MEMORY_WRITE) | | | |
| 273 | { | | | |
| 274 | dst_ptr = (char *) (CPU_RAM + offset); | | | |
| 275 | arc_ptr = buffer; | | | |
| 276 | } | | | |
| 277 | /* | | | |
| 278 | Do the copy | | | |
| 279 | */ | | | |
| 280 | while(number_of_bytes--) | | | |
| 281 | dst_ptr++ = *arc_ptr++; | | | |
| 282 | return(SUCCESS); | | | |
| 283 | } | | | |
| 284 | /* | | | |
| 285 | access CPUreg | | | |
| 286 | */ | | | |
| 287 | At present this sets the PC. | | | |
| 288 | */ | | | |
| 289 | u_short lm_cpureg_access(buffer, offset, number_of_bytes, option, status) | | | |
| 290 | register u_long option; | | | |
| 291 | register u_long offset; | | | |
| 292 | register u_long number_of_bytes; | | | |
| 293 | register char *buffer; | | | |
| 294 | register u_long *status; | | | |
| 295 | { | | | |
| 296 | register void (*execution_address)(); | | | |
| 297 | unsigned long err; | | | |
| 298 | char mod_state; | | | |
| 299 | if(offset >= CPU_RAM_SIZE) | | | |
| 300 | { | | | |
| 301 | *status = NO_MEMORY; | | | |
| 302 | return(FAILURE); | | | |
| 303 | } | | | |
| 304 | /* | | | |
| 305 | set up pointers | | | |
| 306 | */ | | | |
| 307 | if(option == MEMORY_READ) | | | |
| 308 | { | | | |
| 309 | *status = INVALID_PARAMETER; | | | |
| 310 | return(FAILURE); | | | |
| 311 | } | | | |
| 312 | if(option == MEMORY_WRITE) | | | |
| 313 | { | | | |
| 314 | if(lm_svarm_access(&mod_state, MODELER_STATE, sizeof(MODELER_STATE), MEMORY_READ, &err) == FAILURE) | | | |
| 315 | { | | | |
| 316 | sys_out("\nFailed to read NVarM state"); | | | |
| 317 | } | | | |
| 318 | if(mod_state != BOOTED) | | | |
| 319 | { | | | |
| 320 | (void)disable_cache(); | | | |
| 321 | execution_address = (void(*)())offset; | | | |
| 322 | (*execution_address)(); | | | |
| 323 | } | | | |
| 324 | return(FAILURE); | | | |
| 325 | } | | | |
| 326 | /* | | | |
| 327 | access DUART A | | | |
| 328 | */ | | | |
| 329 | At present this sets the DUART. | | | |
| 330 | */ | | | |
| 331 | u_short lm_duart_access(buffer, offset, number_of_bytes, option, status) | | | |
| 332 | register u_long option; | | | |
| 333 | register u_long offset; | | | |
| 334 | register u_long number_of_bytes; | | | |
| 335 | register char *buffer; | | | |
| 336 | register u_long *status; | | | |
| 337 | { | | | |
| 338 | if(number_of_bytes != 1) | | | |
| 339 | { | | | |
| 340 | *status = INVALID_PARAMETER; | | | |
| 341 | return(FAILURE); | | | |
| 342 | } | | | |
| 343 | switch(*buffer & 0xf) | | | |
| 344 | { | | | |
| 345 | case BAUD_110: /* Baud rate 110 */ | | | |
| 346 | case BAUD_300: /* Baud rate 300 */ | | | |
| 347 | case BAUD_1200: /* Baud rate 1200 */ | | | |
| 348 | case BAUD_2400: /* Baud rate 2400 */ | | | |
| 349 | case BAUD_4800: /* Baud rate 4800 */ | | | |
| 350 | case BAUD_9600: /* Baud rate 9600 */ | | | |
| 351 | break; | | | |
| 352 | default: | | | |
| 353 | *status = INVALID_PARAMETER; | | | |
| 354 | } | | | |
| 355 | } | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/lm_rd_wr.c | DATE 5/23/89 | PAGE # 4/39 |
|--|--|---------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 361 | return(FAILURE); | | | |
| 362 | /* | | | |
| 363 | /* set up pointers | | | |
| 364 | */ | | | |
| 365 | if(option == MEMORY_READ) | | | |
| 366 | { | | | |
| 367 | *status = INVALID_PARAMETER; | | | |
| 368 | return(FAILURE); | | | |
| 369 | /* | | | |
| 370 | /* set baud rate | | | |
| 371 | */ | | | |
| 372 | if(option == MEMORY_WRITE) | | | |
| 373 | { | | | |
| 374 | CPU_DISABLE_INTERRUPTS; | | | |
| 375 | uart_init("baud"); | | | |
| 376 | CPU_ENABLE_INTERRUPTS; | | | |
| 377 | } | | | |
| 378 | return(SUCCESS); | | | |
| 379 | /* | | | |
| 380 | /* access DUART B | | | |
| 381 | /* At present this sets the DUART. | | | |
| 382 | /* | | | |
| 383 | u_short lm_duart_access(buffer, offset, number_of_bytes, option, status) | | | |
| 384 | register u_long option; | | | |
| 385 | register u_long offset; | | | |
| 386 | register u_long number_of_bytes; | | | |
| 387 | register char *buffer; | | | |
| 388 | register u_long *status; | | | |
| 389 | { | | | |
| 390 | if(number_of_bytes != 1) | | | |
| 391 | { | | | |
| 392 | *status = INVALID_PARAMETER; | | | |
| 393 | return(FAILURE); | | | |
| 394 | } | | | |
| 395 | switch(*buffer & 0xf) | | | |
| 396 | { | | | |
| 397 | case BAUD_110: /* Baud rate 110 */ | | | |
| 398 | case BAUD_300: /* Baud rate 300 */ | | | |
| 399 | case BAUD_1200: /* Baud rate 1200 */ | | | |
| 400 | case BAUD_2400: /* Baud rate 2400 */ | | | |
| 401 | case BAUD_4800: /* Baud rate 4800 */ | | | |
| 402 | case BAUD_9600: /* Baud rate 9600 */ | | | |
| 403 | break; | | | |
| 404 | default: | | | |
| 405 | *status = INVALID_PARAMETER; | | | |
| 406 | return(FAILURE); | | | |
| 407 | } | | | |
| 408 | /* | | | |
| 409 | /* set up pointers | | | |
| 410 | */ | | | |
| 411 | if(option == MEMORY_READ) | | | |
| 412 | { | | | |
| 413 | *status = INVALID_PARAMETER; | | | |
| 414 | return(FAILURE); | | | |
| 415 | /* | | | |
| 416 | /* set baud rate | | | |
| 417 | */ | | | |
| 418 | if(option == MEMORY_WRITE) | | | |
| 419 | { | | | |
| 420 | CPU_DISABLE_INTERRUPTS; | | | |
| 421 | uart_init("baud"); | | | |
| 422 | CPU_ENABLE_INTERRUPTS; | | | |
| 423 | } | | | |
| 424 | return(SUCCESS); | | | |
| 425 | /* | | | |
| 426 | /* get vram bauda(baud) | | | |
| 427 | unsigned long *baud; | | | |
| 428 | { | | | |
| 429 | unsigned long err; | | | |
| 430 | if(!vram_access(baud, BAUD_RATEA, sizeof_BAUD_RATEA, | | | |
| 431 | MEMORY_READ, &err) == SUCCESS) { | | | |
| 432 | *baud >= 24; | | | |
| 433 | } else { | | | |
| 434 | *baud = BAUD_96_12_24; | | | |
| 435 | } | | | |
| 436 | /* | | | |
| 437 | /* get vram baudb(baud) | | | |
| 438 | unsigned long *baud; | | | |
| 439 | { | | | |
| 440 | unsigned long err; | | | |
| 441 | if(!vram_access(baud, BAUD_RATEB, sizeof_BAUD_RATEB, | | | |
| 442 | MEMORY_READ, &err) == SUCCESS) { | | | |
| 443 | *baud >= 24; | | | |
| 444 | } else { | | | |
| 445 | *baud = BAUD_24_96_12; | | | |
| 446 | } | | | |
| 447 | /* | | | |
| 448 | /* | | | |
| 449 | /* | | | |
| 450 | /* | | | |
| 451 | /* | | | |
| 452 | /* | | | |
| 453 | /* | | | |
| 454 | /* | | | |
| 455 | /* | | | |
| 456 | /* | | | |
| 457 | /* | | | |

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

os/malloc.c

DATE 5/23/89

PAGE #

TIME 4:42:14 pm

1/40

```

1  /* SCCS_ID: malloc.c rev 3.1, 4/24/89 at 07:47:00 */
2
3  /*
4   * Allocator/Deallocator
5   * singly-linked, first-fit
6   * roving pointer, coalescing during allocation
7   * free call resets roving pointer to newly-freed block
8   */
9
10 #define CHECK
11 #define ACCOUNT
12 #ifdef ACCOUNT
13 extern unsigned long available_malloc_size; /* to track remaining memory */
14 #endif ACCOUNT
15 extern int malloc_semaphore;
16 static int err;
17 static unsigned long rover = 0;
18
19 void
20 create_malloc_partition ( start, size )
21 register char *start;
22 register unsigned long size;
23 { register unsigned long addr, end_addr;
24 #ifdef CHECK
25     if ( rover )
26         ( (void) printf ( "create called more than once\n" ) );
27     return;
28 #endif CHECK
29 #ifdef DEBUG
30     (void) printf ( "init(0lx,0lu)\n", ( unsigned long ) start, size );
31 #endif DEBUG
32     addr = ( unsigned long ) start; /* get the start address */
33     end_addr = addr + size; /* point past the arena */
34     addr = addr & 3; /* to round up to 4-byte boundary */
35     end_addr = end_addr & 3; /* to round down to 4-byte boundary */
36     rover = ( unsigned long ) end_addr; /* point past the arena */
37     rover [ 1 ] = addr + 5; /* zero-length last block is busy */
38     rover = ( unsigned long ) addr; /* initialize rover to first block-1 */
39     /*rover = end_addr; /* link first block to last block */
40 } /* create_malloc_partition */
41
42 static char *
43 allocate ( size )
44 register unsigned long size;
45 { register unsigned long save, *next; /* to save our starting point */
46   register unsigned long tmp_size; /* temp to store a blocks size */
47 #ifdef CHECK
48     if ( rover == 0 )
49         return printf ( "allocate called before create\n" ), ( char * ) 0;
50     if ( size == 0 )
51         return printf ( "allocate called with zero size\n" ), ( char * ) 0;
52 #endif CHECK
53 #ifdef DEBUG
54     (void) printf ( "allo(0lu)= ", size );
55 #endif DEBUG
56     size = size & 3; /* to round up to a multiple of 4 */
57     size = size & 3; /* clear bottom 2 bits */
58     save = rover; /* so we know where we have seen it all */
59     do /* loop through blocks */
60     {
61         if ( rover [ -1 ] & 1 ) /* block is busy */
62             rover = ( unsigned long ) ( rover [ -1 ] - 1 ); /* to next block */
63         else /* got a free block */
64             if ( ( unsigned long ) rover [ -1 ] /* check for coalescing */
65                 while ( ( next [ -1 ] & 1 ) == 0 ) /* repeatedly coalesce */
66                     { rover [ -1 ] = next [ -1 ]; /* do the coalesce */
67                       next = ( unsigned long ) rover [ -1 ]; /* check next block */
68                     }
69             tmp_size =
70                 rover [ -1 ] - ( unsigned long ) rover - 4; /* size of block */
71             if ( tmp_size == size ) /* the block is exactly the right size */
72                 { save = rover; /* save the pointer to be returned */
73                   rover = ( unsigned long ) rover [ -1 ]; /* to next block */
74                   ++save [ -1 ]; /* mark this block as busy */
75                 }
76 #ifdef ACCOUNT
77             available_malloc_size -= size & 4;
78 #endif ACCOUNT
79 #ifdef DEBUG
80             (void) printf ( "[1]0lx\n", ( unsigned long ) save );
81 #endif DEBUG
82             return ( char * ) save; /* there's the block requested */
83         else if ( tmp_size > size ) /* the block is larger than we need */
84             { size >>= 2; /* change from a byte count to a word count */
85               rover [ size ] = rover [ -1 ]; /* create next block */
86               rover [ -1 ] = ( unsigned long ) ( rover + tmp_size ); /* link */
87               save = rover; /* save the pointer to be returned */
88               rover += size; /* bump rover to next block */
89               ++save [ -1 ]; /* mark this block as busy */
90             }
91 #ifdef ACCOUNT
92             available_malloc_size -= size << 2;
93 #endif ACCOUNT
94 #ifdef DEBUG
95             (void) printf ( "[2]0lx\n", ( unsigned long ) save );
96 #endif DEBUG
97             return ( char * ) save; /* there's the block requested */
98         else /* the block is too small */
99             rover = ( unsigned long ) rover [ -1 ]; /* to next block */
100     }
101     while ( rover != save ); /* until we come back to where we started */
102 #ifdef DEBUG
103     (void) printf ( "[0]NULL\n" );
104 #endif DEBUG
105 #ifdef DEBUG
106     return 0; /* failure, no free blocks are large enough */
107 #endif DEBUG
108 } /* allocate */
109
110 static void
111 deallocate ( ptr )
112 register char *ptr;
113 {
114 #ifdef CHECK
115     register unsigned long addr;
116     if ( ptr == 0 )
117         ( (void) printf ( "NULL pointer passed to deallocate\n" ) );
118     return;
119 }
120     addr = ( unsigned long ) ptr;

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/malloc.c

DATE 5/23/89 PAGE #
TIME 4:42:14 pm 2/41

SOURCE TEXT

```

121 if ( addr < 1 )
122 { ( void ) printf ( "bad pointer passed to deallocate\n" ) ;
123 return ;
124 }
125 rover = ( unsigned long * ) addr ;
126 if ( ( rover [ -1 ] & 1 ) == 0 )
127 { ( void ) printf ( "free block passed to deallocate\n" ) ;
128 return ;
129 }
130 #endif CHECK
131 #ifdef DEBLOG
132 { ( void ) printf ( "free(1k)\n" , ( unsigned long ) ptr ) ;
133 }
134 #endif DEBLOG
135 rover = ( unsigned long * ) ptr ; /* turn char pointer into long pointer */
136 --rover [ -1 ] ; /* mark the passed block as free */
137 #ifdef ACCOUNT
138 available_malloc_size -- rover [ -1 ] - ( unsigned long ) rover ;
139 #endif ACCOUNT
140 } /* deallocate */
141
142 char *
143 malloc ( size )
144 register unsigned long size ;
145 { register char *ret ;
146 if ( ac_spend ( malloc_semaphore , 0 , terr ) , err )
147 return printf ( "ac_spend() failed in malloc()\n" ) , ( char * ) 0 ;
148 ret = allocate ( size ) ;
149 if ( ac_spend ( malloc_semaphore , terr ) , err )
150 { ( void ) printf ( "ac_spend() failed in malloc()\n" ) ;
151 return ret ;
152 } /* malloc */
153
154 void
155 free ( ptr )
156 register char *ptr ;
157 { if ( ac_spend ( malloc_semaphore , 0 , terr ) , err )
158 { ( void ) printf ( "ac_spend() failed in free()\n" ) ;
159 return ;
160 }
161 deallocate ( ptr ) ;
162 if ( ac_spend ( malloc_semaphore , terr ) , err )
163 { ( void ) printf ( "ac_spend() failed in free()\n" ) ;
164 } /* free */
165
166 char *
167 realloc ( ptr , size )
168 register char *ptr ;
169 register unsigned long size ;
170 { register char *new , *ret ;
171 register unsigned long old ;
172 register unsigned long rover ;
173 #ifdef CHECK
174 if ( ptr == 0 )
175 { ( void ) printf ( "NULL pointer passed to realloc()\n" ) ;
176 return 0 ;
177 }
178 #endif CHECK
179 if ( ac_spend ( malloc_semaphore , 0 , terr ) , err )
180 return printf ( "ac_spend() failed in realloc()\n" ) , ( char * ) 0 ;
181 rover = ( unsigned long * ) ptr ;
182 old = rover [ -1 ] - ( unsigned long ) ptr - 5 ;
183 deallocate ( ptr ) ;
184 ret = new = allocate ( size ) ;
185 if ( new == 0 )
186 { ++rover [ -1 ] ;
187 #ifdef ACCOUNT
188 available_malloc_size -- old + 4 ;
189 #endif ACCOUNT
190 }
191 else if ( new != ptr )
192 { if ( old > size )
193 old = size ;
194 while ( old-- )
195 *new++ = *ptr++ ;
196 if ( ac_spend ( malloc_semaphore , terr ) , err )
197 { ( void ) printf ( "ac_spend() failed in realloc()\n" ) ;
198 return ret ;
199 } /* realloc */
200
201 char *
202 calloc ( count , size )
203 register unsigned long count , size ;
204 { register char *ret ;
205 ret = malloc ( count * size ) ;
206 if ( ret )
207 bzero ( ret , count ) ;
208 return ret ;
209 } /* calloc */
210

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/mem_acc.c | DATE 5/23/89 | PAGE # 1/42 |
|--|---|--------------------------------|-----------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCCS ID: mem_acc.c rev 1.1, 4/24/89 at 07:47:04 */ | | | |
| 2 | | | | |
| 3 | #include "common.h" | | | |
| 4 | #include "lm_rd_wr.h" | | | |
| 5 | #include "device.h" | | | |
| 6 | #include "message.h" | | | |
| 7 | #include "hardware.h" | | | |
| 8 | #include "eeprom.h" | | | |
| 9 | #include "laser.h" | | | |
| 10 | #include "protans.h" | | | |
| 11 | #include "lsetwork.h" | | | |
| 12 | #include "network.h" | | | |
| 13 | | | | |
| 14 | char read_err_buf[] = "Failure to read from the Modeler"; | | | |
| 15 | char write_err_buf[] = "Failure to write to the Modeler"; | | | |
| 16 | | | | |
| 17 | void | | | |
| 18 | process_lm_read(user) | | | |
| 19 | USER_INFO *user; | | | |
| 20 | { | | | |
| 21 | #ifdef MODELER | | | |
| 22 | register u_long memory_type; | | | |
| 23 | register u_long ee_number; | | | |
| 24 | register u_long offset; | | | |
| 25 | register u_long number_of_bytes, i; | | | |
| 26 | register char *err_msg; | | | |
| 27 | char buffer[MAX_RD_WB_BUFFER_SIZE]; | | | |
| 28 | register char *buf_ptr = buffer; | | | |
| 29 | u_long status; | | | |
| 30 | | | | |
| 31 | reset_obuf(); | | | |
| 32 | lm_put_int(READ_ANS); | | | |
| 33 | memory_type = lm_get_int(); | | | |
| 34 | ee_number = lm_get_int(); | | | |
| 35 | offset = lm_get_int(); | | | |
| 36 | number_of_bytes = lm_get_int(); | | | |
| 37 | /* | | | |
| 38 | ** read memory, and copy output buffer | | | |
| 39 | */ | | | |
| 40 | if(proc_lm_rd(memory_type, ee_number, offset, number_of_bytes, buffer, &status) == FAILURE) | | | |
| 41 | { | | | |
| 42 | lm_put_int(1); /* Failure */ | | | |
| 43 | lm_put_char(ERROR_MSG); | | | |
| 44 | err_msg = read_err_buf; | | | |
| 45 | while(*err_msg) | | | |
| 46 | lm_put_char(*err_msg++); | | | |
| 47 | lm_put_char(0); | | | |
| 48 | } | | | |
| 49 | else | | | |
| 50 | { | | | |
| 51 | status=0; | | | |
| 52 | lm_put_int(status); | | | |
| 53 | for(i=0; i < number_of_bytes; ++i) | | | |
| 54 | lm_put_char(*buf_ptr++); | | | |
| 55 | } | | | |
| 56 | end_put(user->id); | | | |
| 57 | return; | | | |
| 58 | #else | | | |
| 59 | | | | |
| 60 | reset_obuf(); | | | |
| 61 | lm_put_int(READ_ANS); | | | |
| 62 | lm_put_int(0); /* status */ | | | |
| 63 | lm_put_int((u_long) FAILURE); | | | |
| 64 | end_put(user->id); | | | |
| 65 | return; | | | |
| 66 | #endif | | | |
| 67 | { | | | |
| 68 | | | | |
| 69 | | | | |
| 70 | void | | | |
| 71 | process_lm_write(user) | | | |
| 72 | USER_INFO *user; | | | |
| 73 | { | | | |
| 74 | #ifdef MODELER | | | |
| 75 | | | | |
| 76 | register u_long memory_type; | | | |
| 77 | register u_long ee_number; | | | |
| 78 | register u_long offset; | | | |
| 79 | register char *err_msg; | | | |
| 80 | register u_long number_of_bytes, i; | | | |
| 81 | register char buffer[MAX_RD_WB_BUFFER_SIZE]; | | | |
| 82 | register char *buf_ptr = buffer; | | | |
| 83 | u_long status; | | | |
| 84 | | | | |
| 85 | reset_obuf(); | | | |
| 86 | lm_put_int(WRITE_ANS); | | | |
| 87 | | | | |
| 88 | memory_type = lm_get_int(); | | | |
| 89 | ee_number = lm_get_int(); | | | |
| 90 | offset = lm_get_int(); | | | |
| 91 | number_of_bytes = lm_get_int(); | | | |
| 92 | buf_ptr = buffer; | | | |
| 93 | for(i=0; i < number_of_bytes; ++i) | | | |
| 94 | *buf_ptr++ = lm_get_char(); | | | |
| 95 | /* | | | |
| 96 | ** write to memory, and copy status to output | | | |
| 97 | */ | | | |
| 98 | if(proc_lm_wr(memory_type, ee_number, offset, number_of_bytes, buffer, &status) == FAILURE) | | | |
| 99 | { | | | |
| 100 | lm_put_int(1); /* Failure */ | | | |
| 101 | lm_put_char(ERROR_MSG); | | | |
| 102 | err_msg = write_err_buf; | | | |
| 103 | while(*err_msg) | | | |
| 104 | lm_put_char(*err_msg++); | | | |
| 105 | lm_put_char(0); | | | |
| 106 | } | | | |
| 107 | else | | | |
| 108 | { | | | |
| 109 | lm_put_int(0); | | | |
| 110 | } | | | |
| 111 | end_put(user->id); | | | |
| 112 | return; | | | |
| 113 | #else | | | |
| 114 | | | | |
| 115 | reset_obuf(); | | | |
| 116 | lm_put_int(WRITE_ANS); | | | |
| 117 | lm_put_int(0); /* status */ | | | |
| 118 | lm_put_int((u_long) FAILURE); | | | |
| 119 | end_put(user->id); | | | |
| 120 | return; | | | |

1819

5,353,243

1820

| | | | | | |
|--|-------|--------------------------------|--|-----------------|--------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/mem_acc.c | | DATE * 5/23/89 | PAGE # |
| | | | | TIME 4:42:15 pm | 2/43 |
| SOURCE TEXT | | | | | |
| LINE # | | | | | |
| 121 | endif | | | | |
| 122 | | | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/mod_err.c | DATE 5/23/89 TIME 4:42:15 pm | PAGE # 1/44 |
|--|---|--------------------------------|---------------------------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCCS ID: mod_err.c rev 3.1.1, 4/24/89 at 07:47:08 */ | | | |
| 2 | #include "common.h" | | | |
| 3 | #include "tag.h" | | | |
| 4 | #include "pac.h" | | | |
| 5 | #include "magic.h" | | | |
| 6 | #include "mod_def.h" | | | |
| 7 | #include "pel.h" | | | |
| 8 | #include "mod_err.h" | | | |
| 9 | #include "cpu.h" | | | |
| 10 | #include "sram.h" | | | |
| 11 | #include "la_rd_wr.h" | | | |
| 12 | #include "vtr.h" | | | |
| 13 | char buffer[256]; | | | |
| 14 | LM_HARDWARE_ERROR modeler_error; | | | |
| 15 | PAC *pacptr[] = { | | | |
| 16 | (PAC *) (LANE_A_OFFSET + PAC_REG_OFFSET), | | | |
| 17 | (PAC *) (LANE_B_OFFSET + PAC_REG_OFFSET), | | | |
| 18 | (PAC *) (LANE_C_OFFSET + PAC_REG_OFFSET), | | | |
| 19 | (PAC *) (LANE_D_OFFSET + PAC_REG_OFFSET) | | | |
| 20 | }; | | | |
| 21 | u_char play_completed_flag = 0; | | | |
| 22 | extern u_short la_read_probe(), la_sram_access(); | | | |
| 23 | extern void reset_cpu(), output_routine(); | | | |
| 24 | void disable_mod_err(); | | | |
| 25 | extern TMC *tmcptr; | | | |
| 26 | extern u_char post_end_of_play; | | | |
| 27 | extern int play_semaphore; | | | |
| 28 | static char unknown_source_of_interrupt = 0; | | | |
| 29 | void | | | |
| 30 | mod_error_isr() | | | |
| 31 | { | | | |
| 32 | unsigned long error; | | | |
| 33 | int err; | | | |
| 34 | PEL pel_access_reg; | | | |
| 35 | PEL *pel_access_reg_ptr; | | | |
| 36 | u_short junk, j, i; | | | |
| 37 | u_short k, lanes_enable, lanes_short, multi_lanes; | | | |
| 38 | register char fatal = 0, found_source_of_interrupt = 0; | | | |
| 39 | u_char found_pel_error; | | | |
| 40 | /* | | | |
| 41 | ** initialize some variables. | | | |
| 42 | lanes_enable = lanes_short = 0; | | | |
| 43 | play_completed_flag; | | | |
| 44 | /* | | | |
| 45 | ** bump count of unknown errors, then set to 0 if source is found. | | | |
| 46 | if (tmcptr -> pattern_intr_enable == 1 && post_end_of_play == 1) | | | |
| 47 | { | | | |
| 48 | ac_sport(play_semaphore, ierr); | | | |
| 49 | if(err != VTRX_OK) | | | |
| 50 | { | | | |
| 51 | printf("Error posting in mod_error_isr() status=%x\n",err); | | | |
| 52 | } | | | |
| 53 | } | | | |
| 54 | /* | | | |
| 55 | ** did we get a error on the lanes | | | |
| 56 | if(tmcptr -> lane_intr == 0) | | | |
| 57 | { | | | |
| 58 | if(tmcptr->backplane_mode == 0 && tmcptr -> pattern_intr_enable == 1) | | | |
| 59 | { | | | |
| 60 | /* | | | |
| 61 | ** This happens if we get EOP without errors | | | |
| 62 | printf("EOP interrupt without any lanes having fault\n"); | | | |
| 63 | /* | | | |
| 64 | tmcptr -> pattern_intr_enable = 0; | | | |
| 65 | /* | | | |
| 66 | ** EOP interrupt. | | | |
| 67 | /* | | | |
| 68 | found_source_of_interrupt=1; | | | |
| 69 | /* | | | |
| 70 | ** we know source. | | | |
| 71 | /* | | | |
| 72 | unknown_source_of_interrupt = 0; | | | |
| 73 | return; | | | |
| 74 | } | | | |
| 75 | } | | | |
| 76 | else { | | | |
| 77 | /* | | | |
| 78 | output_routine("SE\n"); | | | |
| 79 | /* | | | |
| 80 | modeler_error.error = 1; | | | |
| 81 | } | | | |
| 82 | /* | | | |
| 83 | ** see if TMC is driving ERROR | | | |
| 84 | if(tmcptr -> tmc_intr tmcptr -> backplane_error) | | | |
| 85 | { | | | |
| 86 | /*output_routine("TMC error interrupt\n");*/ | | | |
| 87 | modeler_error.tmc_error = 1; | | | |
| 88 | /* | | | |
| 89 | ** if tmc error, read and store tmc error latches | | | |
| 90 | /* | | | |
| 91 | if(tmcptr -> tmc_intr == 1) | | | |
| 92 | { | | | |
| 93 | /* | | | |
| 94 | ** read and store tmc error latches | | | |
| 95 | /* | | | |
| 96 | modeler_error.lane_enable = tmcptr -> lane_enable; | | | |
| 97 | /* | | | |
| 98 | ** determine if it was a multi lane play | | | |
| 99 | ** since shorts can cause sync problems with multi lane | | | |
| 100 | ** play. | | | |
| 101 | /* | | | |
| 102 | lanes_enable = (u_short) modeler_error.lane_enable; | | | |
| 103 | multi_lanes = lanes_enable >> 1; | | | |
| 104 | multi_lanes += (lanes_enable >> 1) & 1; | | | |
| 105 | multi_lanes += (lanes_enable >> 2) & 1; | | | |
| 106 | multi_lanes += (lanes_enable >> 3) & 1; | | | |
| 107 | modeler_error.lane_b_pel_control = tmcptr -> lane_b_pel_control; | | | |
| 108 | modeler_error.lane_b_data_valid = tmcptr -> lane_b_data_valid; | | | |
| 109 | modeler_error.lane_a_pel_control = tmcptr -> lane_a_pel_control; | | | |
| 110 | modeler_error.lane_a_data_valid = tmcptr -> lane_a_data_valid; | | | |
| 111 | modeler_error.lane_d_pel_control = tmcptr -> lane_d_pel_control; | | | |
| 112 | modeler_error.lane_d_data_valid = tmcptr -> lane_d_data_valid; | | | |
| 113 | /* | | | |
| 114 | ** read and store tmc error latches | | | |
| 115 | /* | | | |
| 116 | modeler_error.lane_enable = tmcptr -> lane_enable; | | | |
| 117 | multi_lanes = lanes_enable >> 1; | | | |
| 118 | multi_lanes += (lanes_enable >> 1) & 1; | | | |
| 119 | multi_lanes += (lanes_enable >> 2) & 1; | | | |
| 120 | multi_lanes += (lanes_enable >> 3) & 1; | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/mod_err.c

| | | |
|------|------------|--------|
| DATE | 5/23/89 | PAGE # |
| TIME | 4:42:15 pm | 2/45 |

SOURCE TEXT

```

121 modeler_error.lase_c_pel_control = tmgptr -> lase_c_pel_control;
122 modeler_error.lase_c_data_valid = tmgptr -> lase_c_data_valid;
123 }
124 tmgptr -> backplane_error = 0;
125 tmgptr -> lase_intr_clearl = 0;
126
127 if (lm_tmg_clocken() == FAILURE)
128     reset_cpu_on_fatal_error();
129
130 while( i++ < 0x8000 );
131
132 if (lm_tmg_clockoff() == FAILURE)
133     reset_cpu_on_fatal_error();
134
135 tmgptr -> tmg_intr_clearl = 1;
136
137 /*
138 * sprintf(buffer, "1: %d %x\n",
139 *   tmgptr->tmg_intr, tmgptr->lase_intr);
140 * output_routine(buffer);
141 */
142 /*
143 * which lanes have error
144 */
145 modeler_error.lase_errors |= tmgptr -> lase_intr;
146 /*
147 * this is a fatal condition
148 */
149 fatal = 1;
150 /*
151 * found an error
152 */
153 found_source_of_interrupt=1;
154
155 }
156 i = 0;
157 /*
158 * Check to see if we are
159 * stuck in presentation mode
160 */
161 while( tmgptr -> backplane_mode == 1 )
162 {
163     /*
164     * delay a while before trying
165     * give up if we can not resolve in two tries.
166     */
167     while( i++ < 0x8000 );
168     /*
169     * go about play
170     */
171     lm_tmg_abort_play();
172     if( i++ > 0x8002 at tmgptr -> backplane_mode == 1 )
173     {
174         modeler_error.tmg_error = 1;
175         /*
176         * save in NVRam
177         */
178         (void)lm_nvrw_access((char *) &modeler_error, HARDWARE_ERROR, sizeof(HARDWARE_ERROR), MEMORY_WRITE,&error);
179         reset_cpu(BACKPLANE_ERROR);
180     }
181     /*
182     * found an error
183     */
184     found_source_of_interrupt=1;
185 }
186
187 /*
188 * sprintf(buffer, "2: %d %x\n",
189 *   tmgptr->tmg_intr, tmgptr->lase_intr);
190 * output_routine(buffer);
191 */
192 /*
193 * which lanes have error
194 */
195 modeler_error.lase_errors |= tmgptr -> lase_intr;
196 /*
197 * go check PAC and PEL
198 * PAC first
199 */
200 for( i=0; i != 4; i++)
201 {
202     /*
203     * is there any hardware there
204     */
205     if(! ( tmgptr -> lase_intr & ( 1 << i )))
206         continue;
207     if(lm_read_probe( (u_long *) (pacptr[ i ] ) ) == SUCCESS )
208     {
209         /*
210         * This is fatal refresh error
211         */
212         if( ( pacptr[ i ] -> refresh_error == 1 ) )
213         {
214             modeler_error.pac_lase_errors |= 1 << i;
215             modeler_error.pac_error[ i ].pac_refresh_error = 1;
216             fatal = 1;
217             /*output_routine("refresh error\n");*/
218         }
219         /*
220         * This is fatal request error
221         */
222         if( ( pacptr[ i ] -> request_error == 1 ) )
223         {
224             modeler_error.pac_lase_errors |= 1 << i;
225             modeler_error.pac_error[ i ].pac_request_error = 1;
226             fatal = 1;
227             /*output_routine("request error\n");*/
228         }
229         /*
230         * This is fatal pattern error
231         */
232         if( ( pacptr[ i ] -> pattern_error == 1 ) )
233         {
234             modeler_error.pac_lase_errors |= 1 << i;
235             modeler_error.pac_error[ i ].pac_pattern_error = 1;
236             fatal = 1;
237             /*output_routine("pattern error\n");*/
238         }
239         /*
240         * This is a fatal parity error

```

| Copyright 1989 Logic Modeling Systems | SOURCE PROGRAM os/mod_err.c | # DATE 5/23/89 TIME 4:42:15 pm | PAGE # 3/46 |
|--|--|--------------------------------------|----------------|
| LINE # | SOURCE TEXT | | |
| 241 | if(pacptr[i] -> parity_error == 1) | | |
| 242 | { | | |
| 243 | /* | | |
| 244 | ** The block where error was found | | |
| 245 | ** and branch address | | |
| 246 | */ | | |
| 247 | modeler_error.pac_lane_errors = 1 << i; | | |
| 248 | modeler_error.pac_error[i].pac_branch_address = pacptr[i] -> branch_address; | | |
| 249 | modeler_error.pac_error[i].pac_block_offset = pacptr[i] -> block_offset; | | |
| 250 | modeler_error.pac_error[i].pac_control_word_parity_error = 1; | | |
| 251 | total = 1; | | |
| 252 | /*output routine("parity error\n");*/ | | |
| 253 | } | | |
| 254 | /* | | |
| 255 | ** There was a parity error on high word | | |
| 256 | */ | | |
| 257 | if(pacptr[i] -> high_word_parity_error) | | |
| 258 | { | | |
| 259 | /* | | |
| 260 | ** save the address where parity error was detected | | |
| 261 | */ | | |
| 262 | modeler_error.pac_lane_errors = 1 << i; | | |
| 263 | modeler_error.pac_error[i].pac_parity_error_address = pacptr[i] -> parity_error_address; | | |
| 264 | modeler_error.pac_error[i].pac_high_word_parity_error = 1; | | |
| 265 | total = 1; | | |
| 266 | /*output routine(" High word parity error\n");*/ | | |
| 267 | } | | |
| 268 | /* | | |
| 269 | ** There was a parity error on low word | | |
| 270 | */ | | |
| 271 | if(pacptr[i] -> low_word_parity_error) | | |
| 272 | { | | |
| 273 | /* | | |
| 274 | ** save the address where parity error was detected | | |
| 275 | */ | | |
| 276 | modeler_error.pac_lane_errors = 1 << i; | | |
| 277 | modeler_error.pac_error[i].pac_parity_error_address = pacptr[i] -> parity_error_address; | | |
| 278 | modeler_error.pac_error[i].pac_low_word_parity_error = 1; | | |
| 279 | total = 1; | | |
| 280 | /*output routine(" Low word parity error\n");*/ | | |
| 281 | } | | |
| 282 | /* | | |
| 283 | ** clear the errors, by writing to any bit in the register | | |
| 284 | */ | | |
| 285 | pacptr[i] -> pattern_error = 0; | | |
| 286 | { | | |
| 287 | for(j = 0; j != 8; j++) | | |
| 288 | { | | |
| 289 | pel_access_reg_ptr = (PEL *) (pel_addr(i, j)); | | |
| 290 | if(!is_read_probe((u_long *) pel_access_reg_ptr) == FAILURE) | | |
| 291 | continue; | | |
| 292 | pel_access_reg.car.reg = (u_short) pel_access_reg_ptr -> car.reg; | | |
| 293 | /* | | |
| 294 | ** Check out the PEL | | |
| 295 | */ | | |
| 296 | found_pel_error = FALSE; | | |
| 297 | if(pel_access_reg.car.bit.error1 == 0) | | |
| 298 | { | | |
| 299 | /* | | |
| 300 | sprintf(buffer, "T00411\n", pel_access_reg.car.reg); | | |
| 301 | outputRoutine(buffer); | | |
| 302 | /* | | |
| 303 | if(pel_access_reg.car.bit.present == 1) | | |
| 304 | { | | |
| 305 | /* | | |
| 306 | ** DAB is present | | |
| 307 | */ | | |
| 308 | if(pel_access_reg.car.bit.active == 0) | | |
| 309 | { | | |
| 310 | /* | | |
| 311 | ** DAB plugged in but not active | | |
| 312 | */ | | |
| 313 | printf("DAB inserted\n"); | | |
| 314 | outputRoutine("I"); | | |
| 315 | /* | | |
| 316 | modeler_error.dab_change = 1; | | |
| 317 | modeler_error.pel_error[i * 8 + j].dab_inserted = 1; | | |
| 318 | found_pel_error = TRUE; | | |
| 319 | } | | |
| 320 | } | | |
| 321 | else | | |
| 322 | { | | |
| 323 | /* | | |
| 324 | ** DAB is absent | | |
| 325 | */ | | |
| 326 | if(pel_access_reg.car.bit.initialize == 1) | | |
| 327 | { | | |
| 328 | /* | | |
| 329 | ** DAB was plugged in & set active | | |
| 330 | ** but, since, then has been removed | | |
| 331 | */ | | |
| 332 | printf("DAB removed\n"); | | |
| 333 | outputRoutine("Y"); | | |
| 334 | /* | | |
| 335 | lanes_short = (u_short) 1 << i; | | |
| 336 | modeler_error.dab_change = 1; | | |
| 337 | modeler_error.pel_error[i * 8 + j].dab_removed = 1; | | |
| 338 | found_pel_error = TRUE; | | |
| 339 | } | | |
| 340 | } | | |
| 341 | /* | | |
| 342 | ** play error, either DAB not present or | | |
| 343 | ** PEL not active. | | |
| 344 | */ | | |
| 345 | if(pel_access_reg.car.bit.play_error1 == 0) | | |
| 346 | { | | |
| 347 | /* | | |
| 348 | output routine("Play error\n");*/ | | |
| 349 | lanes_short = (u_short) 1 << i; | | |
| 350 | modeler_error.pel_error[i * 8 + j].play_error = 1; | | |
| 351 | found_pel_error = TRUE; | | |
| 352 | } | | |
| 353 | if(pel_access_reg.car.bit.magic_error1 == 0) | | |
| 354 | { | | |
| 355 | for(k=0; k < 5; ++k) | | |
| 356 | { | | |
| 357 | /* | | |
| 358 | ** check parity | | |
| 359 | */ | | |
| 360 | { | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM | # | DATE | PAGE # |
|--|---|----------------|---|---------|------------|
| | | os/mod_err.c | | 5/23/89 | 4/47 |
| | | | | TIME | 4:42:15 pm |
| SOURCE TEXT | | | | | |
| LINE # | | | | | |
| 361 | if(pel_access_reg_ptr->magic_chip[k].m.parity_out != 0) | | | | |
| 362 | { | | | | |
| 363 | /*output routine "pel parity error"*/ | | | | |
| 364 | modeler_error.pel_error[i = 8 - j].any_magic_parity = 1; | | | | |
| 365 | modeler_error.pel_error[i = 8 + j].magic_parity_out = 1 << k; | | | | |
| 366 | if(!m_read_probe(u_long *) (pacptr[i])) == SUCCESS) | | | | |
| 367 | { | | | | |
| 368 | modeler_error.pac_error[i].pac_branch_address = pacptr[i] -> branch_address; | | | | |
| 369 | modeler_error.pac_error[i].pac_block_offset = pacptr[i] -> block_offset; | | | | |
| 370 | fatal = 1; | | | | |
| 371 | } | | | | |
| 372 | /* | | | | |
| 373 | ** check shorts | | | | |
| 374 | */ | | | | |
| 375 | if(!junk = pel_access_reg_ptr->magic_chip[k].m.short_sample) != 0) | | | | |
| 376 | { | | | | |
| 377 | /*output routine "pel short error"*/ | | | | |
| 378 | lases_short = (u_short) 1 << i; | | | | |
| 379 | modeler_error.pel_error[i = 8 + j].any_magic_short = 1; | | | | |
| 380 | modeler_error.pel_error[i = 8 + j].magic_short[k] = junk; | | | | |
| 381 | } | | | | |
| 382 | } | | | | |
| 383 | /* | | | | |
| 384 | ** reset MAGIC chip | | | | |
| 385 | */ | | | | |
| 386 | junk = pel_access_reg_ptr->magic_chip[0].m.reset; | | | | |
| 387 | found_pel_error = TRUE; | | | | |
| 388 | { | | | | |
| 389 | /* | | | | |
| 390 | ** do reset PEL | | | | |
| 391 | */ | | | | |
| 392 | if(found_pel_error == TRUE) { | | | | |
| 393 | /* | | | | |
| 394 | sprintf(buffer, "RST: %d ad\n", i, j); | | | | |
| 395 | outputRoutine(buffer); | | | | |
| 396 | /* | | | | |
| 397 | Reset the PEL only if we really find | | | | |
| 398 | the source of interrupt. | | | | |
| 399 | */ | | | | |
| 400 | pel_access_reg_ptr->car.bit.reset=0; | | | | |
| 401 | /* | | | | |
| 402 | found source of error, error with PEL | | | | |
| 403 | */ | | | | |
| 404 | modeler_error.pel_error_list = 1 << (i = 8 + j); | | | | |
| 405 | } | | | | |
| 406 | else { | | | | |
| 407 | /* | | | | |
| 408 | Toggle Reset to clear the PEL error */ | | | | |
| 409 | pel_access_reg_ptr->car.bit.reset=0; | | | | |
| 410 | pel_access_reg_ptr->car.bit.reset=1; | | | | |
| 411 | } | | | | |
| 412 | found_source_of_interrupt=1; | | | | |
| 413 | } | | | | |
| 414 | } | | | | |
| 415 | } | | | | |
| 416 | if(!tagptr -> lase_intr) | | | | |
| 417 | { | | | | |
| 418 | tagptr -> lase_intr_enable = 0; | | | | |
| 419 | tagptr -> lase_intr_enable = 1; | | | | |
| 420 | /* | | | | |
| 421 | no ooe is driving error. | | | | |
| 422 | */ | | | | |
| 423 | found_source_of_interrupt=1; | | | | |
| 424 | } | | | | |
| 425 | if(fatal == 1) | | | | |
| 426 | { | | | | |
| 427 | /* | | | | |
| 428 | found source of error, but it was fatal. | | | | |
| 429 | */ | | | | |
| 430 | found_source_of_interrupt=1; | | | | |
| 431 | if(modeler_error.tag_error == 1 && | | | | |
| 432 | ((lases_enable & lases_short) && multi_lases > 1)) | | | | |
| 433 | modeler_error.tag_error = 0; | | | | |
| 434 | else | | | | |
| 435 | { | | | | |
| 436 | /* | | | | |
| 437 | save in NVars | | | | |
| 438 | */ | | | | |
| 439 | (void)lm_nvars_access((char *) &modeler_error, | | | | |
| 440 | HARDWARE_ERROR, sizeof HARDWARE_ERROR, | | | | |
| 441 | MEMORY_WRITE_ERROR); | | | | |
| 442 | reset_cpu(BACKPLANE_ERROR); | | | | |
| 443 | } | | | | |
| 444 | } | | | | |
| 445 | } | | | | |
| 446 | if(found_source_of_interrupt == 1) | | | | |
| 447 | { | | | | |
| 448 | unknown_source_of_interrupt = 0; | | | | |
| 449 | } | | | | |
| 450 | else | | | | |
| 451 | { | | | | |
| 452 | if(unknown_source_of_interrupt++ > 2) | | | | |
| 453 | { | | | | |
| 454 | (void)disable_mod_err(); | | | | |
| 455 | printf("Unknown : rce of backplane interrupt\n"); | | | | |
| 456 | modeler_error.unknown_source_of_interrupt=1; | | | | |
| 457 | } | | | | |
| 458 | } | | | | |
| 459 | } | | | | |
| 460 | } | | | | |
| 461 | void | | | | |
| 462 | enable_mod_err() | | | | |
| 463 | { | | | | |
| 464 | cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG; | | | | |
| 465 | /* | | | | |
| 466 | printf("enable_mod_err()\n"); | | | | |
| 467 | ** enable clock board interrupts | | | | |
| 468 | */ | | | | |
| 469 | p_cpu_ctl_reg->tag_intr_ena = 1; | | | | |
| 470 | } | | | | |
| 471 | void | | | | |
| 472 | disable_mod_err() | | | | |
| 473 | { | | | | |
| 474 | cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG; | | | | |
| 475 | /* | | | | |
| 476 | printf("disable_mod_err()\n"); | | | | |
| 477 | ** enable clock board interrupts | | | | |
| 478 | */ | | | | |
| 479 | | | | | |

| | | | | |
|--|---|--------------------------------|---------------------------------|----------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/mod_err.c | DATE 5/23/89 TIME 4:42:15 pm | PAGE # 5/48 |
| LINE # | SOURCE TEXT | | | |
| 480 | /* | | | |
| 481 | P_cpu_ctl_reg->tmg_intr_err = 0; | | | |
| 482 | */ | | | |
| 483 | reset_cpu_os_fatal_error() | | | |
| 484 | { | | | |
| 485 | unsigned long error; | | | |
| 486 | /* | | | |
| 487 | ** save in NVRAM | | | |
| 488 | */ | | | |
| 489 | { | | | |
| 490 | (void)lm_nvram_access((char *) &modeler_error, HARDWARE_ERROR, sizeof(HARDWARE_ERROR), MEMORY_WRITE, &error); | | | |
| 491 | reset_cpu(BACKPLANE_ERROR); | | | |
| 492 | } | | | |
| 493 | } | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/nvsram.c

DATE * 5/23/89
TIME 4:42:16 pm

PAGE #
1/49

```

1  // SCCS_ID: nvsram.c rev 3.1, 4/24/89 at 07:47:12
2
3  /*
4  ** Routines to access NVSRAM
5  */
6  #include "common.h"
7  #include "cpu.h"
8  #include "lm_rd_wr.h"
9  #include "mod_err.h"
10 #include "nvsram.h"
11 #include "uart.h"
12
13 char modular_state = 0; /* undefined */
14
15 static char sram_init = 0;
16 char is_valid_nvsram = FALSE;
17
18 /*
19 ** Compute checksum of NVSRAM, passed pointer to start, returns as u_long
20 */
21 u_long
22 nvsram_compute_checksum(sram)
23 u_long *sram;
24 {
25     register u_short i;
26     register u_long checksum;
27
28     checksum = 0;
29     for(i = 0; i < (CPU_NVSRAM_SIZE / 4); i++)
30     {
31         checksum += (*sram + 0xffffffff);
32         sram++;
33     }
34     /*
35     ** Return 2's complement
36     */
37     return (((~checksum) + 0xffffffff) + 0x01000000);
38 }
39
40 u_short
41 lm_nvsram_access(data_struct, field, sizeof_field, option, error)
42 register Char *data_struct;
43 register u_long field;
44 register u_long sizeof_field, option;
45 register u_long *error;
46 {
47     u_short version;
48     register u_long *sram = (u_long *) (CPU_NVSRAM + field);
49     cpu_control_reg_struct *p_cpu_ctrl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG;
50     register u_long checksum;
51     register u_short i;
52     *error = NO_NVSRAM_ERROR;
53
54     /*
55     ** If reading or writing we have valid sram
56     ** and sram access is within bounds
57     */
58     if(option == MEMORY_READ || option == MEMORY_WRITE)
59     {
60         if((field + sizeof_field) > CPU_NVSRAM_SIZE)
61         {
62             *error = NO_NVSRAM;
63             return (FAILURE);
64         }
65         if(!sram_init)
66         {
67             *error = INVALID_NVSRAM;
68             return (FAILURE);
69         }
70     }
71
72     switch (option)
73     {
74     case MEMORY_READ:
75         /*
76         ** perform read
77         */
78         for(i = 0; i < sizeof_field; i++)
79         {
80             *data_struct++ = (u_char) (*sram >> 24);
81             sram++;
82         }
83         return (SUCCESS);
84     case MEMORY_WRITE:
85         /*
86         ** perform write
87         */
88         p_cpu_ctrl_reg->nvsram_write_err = 1;
89         for(i = 0; i < sizeof_field; i++)
90         {
91             *sram++ = (u_long) *data_struct << 24;
92             data_struct++;
93         }
94         /*
95         ** setup pointer again
96         */
97         sram = (u_long *) (CPU_NVSRAM + CHECKSUM);
98         /*
99         ** clear previous checksum, calculate, and store
100        */
101        *sram = 0;
102        checksum = nvsram_compute_checksum(sram);
103        *sram = checksum;
104        p_cpu_ctrl_reg->nvsram_write_err = 0;
105        return (SUCCESS);
106    case MEMORY_INIT:
107        /*
108        ** Initialize sram
109        */
110        p_cpu_ctrl_reg->nvsram_write_err = 1;
111        /*
112        ** write signature
113        */
114        *sram = (u_long *) (CPU_NVSRAM + BSIC);
115        *sram++ = (u_long) '1' << 24;
116        *sram++ = (u_long) 'm' << 24;
117        *sram++ = (u_long) 's' << 24;
118        *sram++ = (u_long) 'i' << 24;
119        /*
120        ** initialize rest of sram to 0

```

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/nvsram.c

#

DATE 5/23/89

PAGE #

TIME 4:42:16 pm

2/50

| LINE # | SOURCE TEXT |
|--------|--|
| 121 | /* |
| 122 | for(i = 4; i < (CPU_NVSRAM_SIZE / 4); i++) |
| 123 | *aram++ = 0; |
| 124 | /* |
| 125 | ** set up BAUD rate. |
| 126 | */ |
| 127 | aram = (u_long *) CPU_NVSRAM; |
| 128 | *(u_long *) ((i*4) aram + BAUD_RATEA) = |
| 129 | (u_long) (BAUD_96_12_24 << 24); |
| 130 | *(u_long *) ((i*4) aram + BAUD_RATEB) = |
| 131 | (u_long) (BAUD_24_96_12 << 24); |
| 132 | /* |
| 133 | ** compute and store checksum |
| 134 | */ |
| 135 | checksum = nvsram_compute_checksum(aram); |
| 136 | *(u_long *) ((i*4) aram + CHECKSUM) = checksum; |
| 137 | */ |
| 138 | aram_init = 1; |
| 139 | p_cpu_ctrl_reg->nvsram_write_ena = 0; |
| 140 | return (SUCCESS); |
| 141 | case MEMORY_VALIDATE: |
| 142 | /* |
| 143 | ** validate NVSRAM signature |
| 144 | */ |
| 145 | aram = (u_long *) (CPU_NVSRAM + NSIG); |
| 146 | /* |
| 147 | ** check for aram initialized, but without a boot |
| 148 | ** structure |
| 149 | */ |
| 150 | if(((aram >> 24) != 'L') |
| 151 | ((aram >> 24) != 'M') |
| 152 | ((aram >> 24) != 'S') |
| 153 | ((aram >> 24) != 'I')) |
| 154 | { |
| 155 | *error = INVALID_NVSRAM; |
| 156 | return (FAILURE); |
| 157 | } |
| 158 | /* |
| 159 | ** calculate checksum and verify |
| 160 | */ |
| 161 | checksum = nvsram_compute_checksum((u_long *) CPU_NVSRAM); |
| 162 | if (checksum != 0) |
| 163 | { |
| 164 | *error = INVALID_NVSRAM; |
| 165 | return (FAILURE); |
| 166 | } |
| 167 | /* |
| 168 | ** validate NVSRAM version. |
| 169 | */ |
| 170 | version = (u_short) *(u_char *) (CPU_NVSRAM + NVSR + 1*4); |
| 171 | if (version < EPROM_NVSRAM_VERSION) { |
| 172 | *error = STALE_HOST_SOFTWARE; |
| 173 | return (FAILURE); |
| 174 | } |
| 175 | else if (version > EPROM_NVSRAM_VERSION) { |
| 176 | *error = STALE_EPROM_SOFTWARE; |
| 177 | return (FAILURE); |
| 178 | } |
| 179 | in_valid_nvsram = TRUE; |
| 180 | return (SUCCESS); |
| 181 | case MEMORY_DIAG_IN_T: |
| 182 | aram_init = 1; |
| 183 | return (SUCCESS); |
| 184 | default: |
| 185 | return (FAILURE); |
| 186 | } |
| 187 | } |
| 188 | } |
| 189 | } |
| 190 | } |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/parity.c | DATE * 5/23/89 TIME 4:42:16 pm | PAGE # 1/51 |
|--|--|-------------------------------|-----------------------------------|----------------|
| LINE # | SOURCE TEXT | | | |
| 1 | /* SCCS ID: parity.c rev 3.1, 4/24/89 at 07:47:16 */ | | | |
| 2 | /* | | | |
| 3 | ** Parity error handler | | | |
| 4 | ** | | | |
| 5 | */ | | | |
| 6 | #include "common.h" | | | |
| 7 | #include "cpu.h" | | | |
| 8 | #include "mod_err.h" | | | |
| 9 | #include "sram.h" | | | |
| 10 | #include "ls_rd_wr.h" | | | |
| 11 | extern u_short ls_sram_access(); | | | |
| 12 | extern void outputRoutine(), reset_cpu(); | | | |
| 13 | parity_error() | | | |
| 14 | { | | | |
| 15 | register cpu_control_reg_struct *control_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG; | | | |
| 16 | register cpu_par_err_reg *parity_reg = (cpu_par_err_reg *) CPU_PAR_ERR_REG, parity, | | | |
| 17 | u_long error; | | | |
| 18 | char buffer[100]; | | | |
| 19 | { | | | |
| 20 | parity = *parity_reg; | | | |
| 21 | if (control_reg->not_parity_intr == 0) | | | |
| 22 | { | | | |
| 23 | sprintf(buffer, "Parity error addr = %08x\n", | | | |
| 24 | parity_reg->error_addr * 4); | | | |
| 25 | outputRoutine(buffer); | | | |
| 26 | sprintf(buffer, "68020 to VME to LANCE to\n", | | | |
| 27 | parity_reg->not_68020_master, | | | |
| 28 | parity_reg->not_VME_master, | | | |
| 29 | parity_reg->not_LANCE_master); | | | |
| 30 | outputRoutine(buffer); | | | |
| 31 | sprintf(buffer, "hi to um to ls to to to\n", | | | |
| 32 | parity_reg->not_error_hi, | | | |
| 33 | parity_reg->not_error_um, | | | |
| 34 | parity_reg->not_error_ls, | | | |
| 35 | parity_reg->not_error_lo); | | | |
| 36 | outputRoutine(buffer); | | | |
| 37 | (void)ls_sram_access((char *) parity_reg, PARITY_ERR, sizeof(PARITY_ERR), MEMORY_WRITE, &error); | | | |
| 38 | control_reg->parity_force = 0; | | | |
| 39 | reset_cpu(PARITY_ERROR); | | | |
| 40 | } | | | |
| 41 | } | | | |
| 42 | | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/timer.c

DATE 5/23/89
TIME 4:42:16 pm

PAGE #
1/52

```

1  // SCCS_ID: timer.c Rev 3.1, 4/24/89 at 07:47:13
2
3  //
4  // Initialize all three timers.
5  //
6  #include "common.h"
7  #include "cpu.h"
8  #include "timer.h"
9
10 init_all_timer()
11 {
12     unsigned long *timer;
13     timer_control = timer_control_reg;
14     timer_control timer_254_control;
15
16     //
17     // set up timer 0
18     //
19     // write the control register
20     // timer 0, write lsb then msb, use binary value for counter.
21     //
22     timer_254_control.select_timer_counter = SELECT_TIMER_COUNTER0;
23     timer_254_control.r_w_timer_counter = R_W_LSB_MSB_TIMER_CNTR;
24     timer_254_control.timer_mode = TIMER_MODE0;
25     timer_254_control.bcd = BINARY;
26     //
27     // set up a pointer to the 254 timer chip, control register
28     // and setup timer 0
29     //
30     timer_control_reg = (timer_control *) (CPU_TIMER + TIMER_CNTL_WORD);
31     timer_control_reg = timer_254_control;
32     //
33     // write LSB then MSB counter regs to obtain 5 ms waveform
34     // ( 0x1400 ) = 1024000 / 200;
35     //
36     timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER0);
37     *timer = 0x00000000;
38     timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER0);
39     *timer = 0x14000000;
40     //
41     // set up timer 1
42     //
43     // write the control register
44     // timer 1, write lsb then msb, use binary value for counter.
45     //
46     timer_254_control.select_timer_counter = SELECT_TIMER_COUNTER1;
47     //
48     // write to the 254 timer chip, control register
49     //
50     timer_control_reg = timer_254_control;
51     //
52     // write LSB then MSB counter regs to obtain 5 ms waveform
53     // 320 ( 0x140 ) = 64000 / 200;
54     //
55     timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER1);
56     *timer = 0x40000000;
57     timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER1);
58     *timer = 0x01000000;
59     //
60     // set up timer 2
61     //
62     // write the control register
63     // set up timer mode 0
64     // This provides tick for a/v clock
65     // timer 2, write lsb then msb, use binary value for counter.
66     //
67     timer_254_control.timer_mode = TIMER_MODE0;
68     timer_254_control.select_timer_counter = SELECT_TIMER_COUNTER2;
69     //
70     // write to the 254 timer chip, control register
71     //
72     timer_control_reg = timer_254_control;
73     //
74     // write LSB then MSB counter regs to obtain 5 ms waveform
75     // 320 ( 0x140 ) = 64000 / 200;
76     //
77     timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER2);
78     *timer = 0x40000000;
79     timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER2);
80     *timer = 0x01000000;
81     //
82     // setup timer 0 to trigger after time microseconds
83     //
84     static unsigned short time_to_wait;
85     void
86     setup_timer0( time )
87     unsigned long time;
88     {
89         unsigned long *timer;
90         timer_control = timer_control_reg;
91         timer_control timer_254_control;
92
93         //
94         // set up timer 0
95         //
96         time_to_wait = time - ((1024000 / ( 1000000 / time)) & 0xffff)-1;
97         //
98         // write the control register
99         // timer 0, write lsb then msb, use binary value for counter.
100        //
101        timer_254_control.select_timer_counter = SELECT_TIMER_COUNTER0;
102        timer_254_control.r_w_timer_counter = R_W_LSB_MSB_TIMER_CNTR;
103        timer_254_control.timer_mode = TIMER_MODE0;
104        timer_254_control.bcd = BINARY;
105        //
106        // set up a pointer to the 254 timer chip, control register
107        // and setup timer 0
108        //
109        timer_control_reg = (timer_control *) (CPU_TIMER + TIMER_CNTL_WORD);
110        timer_control_reg = timer_254_control;
111        //
112        // write LSB then MSB counter regs
113        //
114        timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER0);
115        *timer = time << 24;
116        timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER0);
117    }
118
119
120

```

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/timer.c | DATE * 5/23/89 | PAGE # 2/53 |
|--|--|---|-------------------|----------------|
| LINE # | | SOURCE TEXT | | |
| 121 | | *timer = time << 16; | | |
| 122 | | } | | |
| 123 | | /* | | |
| 124 | | ** returns FAILURE if timer0 has not expired | | |
| 125 | | ** else SUCCESS | | |
| 126 | | ** must call setup_timer0(microseconds_to_time); | | |
| 127 | | ** before checking. | | |
| 128 | | ** | | |
| 129 | | unsigned short | | |
| 130 | | read_timer0(elapsed); | | |
| 131 | | u_long *elapsed; | | |
| 132 | | { | | |
| 133 | | counter_status timer_0_status, *ptr_timer_0_status = (counter_status *) (CPU_TIMER + TIMER_COUNTER0); | | |
| 134 | | timer_status *timer_status_reg; | | |
| 135 | | timer_status timer_8254_status; | | |
| 136 | | unsigned long *timer; | | |
| 137 | | u_short time; | | |
| 138 | | /* | | |
| 139 | | ** read timer 0 status | | |
| 140 | | ** The bits are not the same | | |
| 141 | | */ | | |
| 142 | | timer_8254_status.read_back = READ_BACK_TIMER; /* READ BACK COMMAND */ | | |
| 143 | | timer_8254_status.count_status = READ_BACK_STATUS; /* read status */ | | |
| 144 | | timer_8254_status.counter = READ_TIMER0; /* timer 0 */ | | |
| 145 | | timer_8254_status.zero = 0; /* must be 0 */ | | |
| 146 | | /* | | |
| 147 | | ** set up a pointer to the 8254 timer chip, control register | | |
| 148 | | ** and setup timer 0 | | |
| 149 | | */ | | |
| 150 | | timer_status_reg = (timer_status *) (CPU_TIMER + TIMER_CNTL_WORD); | | |
| 151 | | *timer_status_reg = timer_8254_status; | | |
| 152 | | timer_0_status = *ptr_timer_0_status; | | |
| 153 | | if(timer_0_status.output == OUT_HIGH) { | | |
| 154 | | return(SUCCESS); | | |
| 155 | | } | | |
| 156 | | else | | |
| 157 | | { | | |
| 158 | | /* | | |
| 159 | | ** read timer 0 count | | |
| 160 | | */ | | |
| 161 | | timer_8254_status.read_back = READ_BACK_TIMER; /* READ BACK COMMAND */ | | |
| 162 | | timer_8254_status.count_status = READ_BACK_COUNT; /* read count */ | | |
| 163 | | timer_8254_status.counter = READ_TIMER0; /* timer 0 */ | | |
| 164 | | timer_8254_status.zero = 0; /* must be 0 */ | | |
| 165 | | *timer_status_reg = timer_8254_status; | | |
| 166 | | timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER0); | | |
| 167 | | time = (u_short)((*timer >> 24) & 0xff); | | |
| 168 | | time = (u_short)((*timer >> 16) & 0xff00); | | |
| 169 | | time = time_to_wait - time; | | |
| 170 | | *elapsed = (u_long)time * 1000 / 1024; | | |
| 171 | | return(FAILURE); | | |
| 172 | | } | | |
| 173 | | } | | |
| 174 | | /* | | |
| 175 | | ** setup timer 1 to trigger after time microseconds | | |
| 176 | | ** | | |
| 177 | | void | | |
| 178 | | setup_timer1(time) | | |
| 179 | | unsigned long time; | | |
| 180 | | { | | |
| 181 | | unsigned long *timer; | | |
| 182 | | timer_control *timer_control_reg; | | |
| 183 | | timer_control timer_8254_control; | | |
| 184 | | /* | | |
| 185 | | ** set up timer 1 | | |
| 186 | | */ | | |
| 187 | | time = ((64000 / (1000000 / time)) & 0xffff)-1; | | |
| 188 | | /* | | |
| 189 | | ** write the control register | | |
| 190 | | ** timer 1, write lsb then msb, use binary value for counter. | | |
| 191 | | */ | | |
| 192 | | timer_8254_control.select_timer_counter = SELECT_TIMER_COUNTER1; | | |
| 193 | | timer_8254_control.r_w_timer_counter = R_W_LSB_MSB_TIMER_CNTR; | | |
| 194 | | timer_8254_control.timer_mode = TIMER_MODE0; | | |
| 195 | | timer_8254_control.bcd = BINARY; | | |
| 196 | | /* | | |
| 197 | | ** set up a pointer to the 8254 timer chip, control register | | |
| 198 | | ** and setup timer 1 | | |
| 199 | | */ | | |
| 200 | | timer_control_reg = (timer_control *) (CPU_TIMER + TIMER_CNTL_WORD); | | |
| 201 | | *timer_control_reg = timer_8254_control; | | |
| 202 | | /* | | |
| 203 | | ** write LSB then MSB counter regs | | |
| 204 | | */ | | |
| 205 | | timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER1); | | |
| 206 | | *timer = time << 24; | | |
| 207 | | timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER1); | | |
| 208 | | *timer = time << 16; | | |
| 209 | | } | | |
| 210 | | /* | | |
| 211 | | ** returns FAILURE if timer0 has not expired | | |
| 212 | | ** else SUCCESS | | |
| 213 | | ** must call setup_timer1(microseconds_to_time); | | |
| 214 | | ** before checking. | | |
| 215 | | ** | | |
| 216 | | unsigned short | | |
| 217 | | read_timer1() | | |
| 218 | | { | | |
| 219 | | counter_status timer_1_status, *ptr_timer_1_status = (counter_status *) (CPU_TIMER + TIMER_COUNTER1); | | |
| 220 | | timer_status *timer_status_reg; | | |
| 221 | | timer_status timer_8254_status; | | |
| 222 | | /* | | |
| 223 | | ** read timer 1 status | | |
| 224 | | ** The bits are not the same | | |
| 225 | | */ | | |
| 226 | | timer_8254_status.read_back = READ_BACK_TIMER; /* READ BACK COMMAND */ | | |
| 227 | | timer_8254_status.count_status = READ_BACK_STATUS; /* read status */ | | |
| 228 | | timer_8254_status.counter = READ_TIMER1; /* timer 1 */ | | |
| 229 | | timer_8254_status.zero = 0; /* must be 0 */ | | |
| 230 | | /* | | |
| 231 | | ** set up a pointer to the 8254 timer chip, control register | | |
| 232 | | ** | | |

| | | | | | |
|--|--|------------------------------|---|-----------------|--------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM os/timer.c | # | DATE > 5/23/89 | PAGE # |
| | | | | TIME 4:42:16 pm | 3/54 |
| LINE # | SOURCE TEXT | | | | |
| 241 | ** and setup timer 1 | | | | |
| 242 | */ | | | | |
| 243 | timer_status_reg = (timer_status *) (CPU_TIMER + TIMER_CNTL_WORD); | | | | |
| 244 | *timer_status_reg = timer_8254_status; | | | | |
| 245 | | | | | |
| 246 | timer_1_status = *ptr_timer_1_status; | | | | |
| 247 | | | | | |
| 248 | if(timer_1_status.output == OUT_HIGH) | | | | |
| 249 | return(SUCCESS); | | | | |
| 250 | else | | | | |
| 251 | { | | | | |
| 252 | return(FAILURE); | | | | |
| 253 | } | | | | |
| 254 | } | | | | |

Copyright 1989
Logic Modeling Systems

SOURCE PROGRAM
os/vrtxkeys.c

| | | | |
|---|------|------------|--------|
| # | DATE | 5/23/89 | PAGE # |
| | TIME | 4:42:16 pm | 1/55 |

```

1  // SCCS ID: vrtxkeys.c rev 1.1, 4/24/89 at 07:47:22
2
3  int key_hit;
4
5  #define GET_KEY 1
6  #define SIZEOFARRAY 0x1f /* MUST ALWAYS BE A POWER OF 2 minus 1 */
7  static char in_ptr = 0, out_ptr = 0, count_keys = 0, array[SIZEOFARRAY+1];
8
9  /*
10  ** buffer up keys, as they arrive
11  ** this is a test and it should be given the highest priority
12  ** count_keys keeps track of keys.
13  */
14  void
15  put_getc()
16  {
17  char ch;
18  long err;
19  while( 1 )
20  {
21      ch = sc_getc();
22      if( count_keys < SIZEOFARRAY )
23      {
24          array[ in_ptr++ ] = ch;
25          in_ptr %= SIZEOFARRAY;
26          count_keys++;
27          count_keys %= SIZEOFARRAY;
28          sc_send( key_hit, &err );
29          if( err )
30              printf( "error posting to semaphore" );
31      }
32  }
33  }
34
35  /*
36  ** stick around till a key is entered
37  */
38  get_key()
39  {
40  int err;
41  char ch;
42  while( 1 )
43  {
44      sc_send( key_hit, 0, &err );
45      if( err )
46          printf( "error pending for semaphore" );
47      if( count_keys )
48      {
49          count_keys--;
50          ch = array[ out_ptr++ ];
51          out_ptr %= SIZEOFARRAY;
52          return( ch );
53      }
54  }
55  }
56
57  /*
58  ** return number of unprocessed keys
59  */
60  check_key()
61  {
62  return( count_keys );
63  }
64
65  }

```


| SOURCE PROGRAM | | DATE | PAGE # |
|----------------|---|------------|--------|
| misc/message.c | | 5/23/89 | 1/1 |
| TIME | | 1:20:42 pm | |
| SOURCE TEXT | | | |
| LINE # | SOURCE TEXT | | |
| 1 | /* SCCS ID: message.c; 5/24/89 at 07:37:13 */ | | |
| 2 | #ifndef MODEL | | |
| 3 | /* On the host, use the definition for... */ | | |
| 4 | #include <stdio.h> | | |
| 5 | #endif | | |
| 6 | #include <errno.h> | | |
| 7 | #include <varargs.h> | | |
| 8 | /* On the modular, use the definition for sprintf() */ | | |
| 9 | #include "common.h" | | |
| 10 | #include "message.h" | | |
| 11 | #endif VMS | | |
| 12 | #include <descrip> | | |
| 13 | #endif | | |
| 14 | #define ABSOLUTE_MAXIMUM_MESSAGE 1024 | | |
| 15 | typedef struct message { | | |
| 16 | u_short message_severity; | | |
| 17 | char message_text; | | |
| 18 | struct message *next_message; | | |
| 19 | } MESSAGE; | | |
| 20 | static u_short error_message_present = FALSE; | | |
| 21 | static u_short warning_message_present = FALSE; | | |
| 22 | static MESSAGE *start_of_message_list = (MESSAGE *) NULL; | | |
| 23 | static MESSAGE *end_of_message_list = (MESSAGE *) NULL; | | |
| 24 | static MESSAGE *out_of_memory; | | |
| 25 | static u_short no_memory_already_linked = FALSE; | | |
| 26 | static char *no_memory_message = "cannot allocate memory to store error/warning message"; | | |
| 27 | static | | |
| 28 | error_count = 0; | | |
| 29 | static | | |
| 30 | warning_count = 0; | | |
| 31 | extern void link_message(); | | |
| 32 | extern void queue_message(); | | |
| 33 | extern void get_system_string(); | | |
| 34 | extern void handle_out_of_memory(); | | |
| 35 | /*VARIABLES*/ | | |
| 36 | void | | |
| 37 | link_message(va_list) | | |
| 38 | { | | |
| 39 | va_list args; | | |
| 40 | u_long type; | | |
| 41 | char *format; | | |
| 42 | int error_number; | | |
| 43 | char system_message[NOX_MESSAGE]; | | |
| 44 | char ln_msg[ABSOLUTE_MAXIMUM_MESSAGE]; | | |
| 45 | va_start(args); | | |
| 46 | type = (u_long) va_arg(args, u_long); | | |
| 47 | if (type == VMS_ERROR_MSG type == VMS_WARNING_MSG) | | |
| 48 | error_number = va_arg(args, int); | | |
| 49 | else if (type == SYS_ERROR_MSG type == SYS_WARNING_MSG) | | |
| 50 | error_number = errno; | | |
| 51 | format = va_arg(args, char *); | | |
| 52 | (void) vfprintf(ln_msg, format, args); | | |
| 53 | va_end(args); | | |
| 54 | if (type == SYS_ERROR_MSG type == SYS_WARNING_MSG | | |
| 55 | type == VMS_ERROR_MSG type == VMS_WARNING_MSG) | | |
| 56 | { | | |
| 57 | get_system_string(error_number, system_message); | | |
| 58 | (void) strcat(ln_msg, system_message); | | |
| 59 | } | | |
| 60 | if (type == ERROR_MSG type == SYS_ERROR_MSG type == VMS_ERROR_MSG) { | | |
| 61 | ++error_count; /* does not work if out of memory */ | | |
| 62 | error_message_present = TRUE; | | |
| 63 | queue_message(ERROR_MSG, ln_msg); | | |
| 64 | } | | |
| 65 | else { | | |
| 66 | ++warning_count; /* does not work if out of memory */ | | |
| 67 | warning_message_present = TRUE; | | |
| 68 | queue_message(WARNING_MSG, ln_msg); | | |
| 69 | } | | |
| 70 | } | | |
| 71 | void | | |
| 72 | link_message_queue() | | |
| 73 | { | | |
| 74 | register MESSAGE *message; | | |
| 75 | error_count = 0; | | |
| 76 | warning_count = 0; | | |
| 77 | error_message_present = FALSE; | | |
| 78 | warning_message_present = FALSE; | | |
| 79 | while(start_of_message_list != (MESSAGE *) NULL) { | | |
| 80 | message = start_of_message_list; | | |
| 81 | start_of_message_list = start_of_message_list->next_message; | | |
| 82 | if (message == out_of_memory) { | | |
| 83 | no_memory_already_linked = FALSE; | | |
| 84 | } | | |
| 85 | else { | | |
| 86 | (void) free(message->message_text); | | |
| 87 | (void) free((char *)message); | | |
| 88 | } | | |
| 89 | } | | |
| 90 | u_short | | |
| 91 | link_message_queue(type, str) | | |
| 92 | u_short *type; | | |
| 93 | char *str; | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM misc/message.c | DATE 5/23/89 | PAGE # 2/2 |
|--|---|----------------------------------|-----------------|---------------|
| LINE # | SOURCE TEXT | | | |
| 21 | { | | | |
| 22 | register MESSAGE *old_message; | | | |
| 23 | if (start_of_message_list == (MESSAGE *) NULL) { | | | |
| 24 | error_message_present = FALSE; | | | |
| 25 | warning_message_present = FALSE; | | | |
| 26 | error_count = 0; | | | |
| 27 | warning_count = 0; | | | |
| 28 | return(FAILURE); | | | |
| 29 | } | | | |
| 30 | *type = start_of_message_list->message_severity; | | | |
| 31 | (void) strcpy(str, start_of_message_list->message_text); | | | |
| 32 | if (*type == ERROR_MSG) | | | |
| 33 | --error_count; | | | |
| 34 | else | | | |
| 35 | --warning_count; | | | |
| 36 | old_message = start_of_message_list; | | | |
| 37 | start_of_message_list = start_of_message_list->next_message; | | | |
| 38 | if (old_message != test_of_memory) { | | | |
| 39 | (void) free(old_message->message_text); | | | |
| 40 | (void) free((char *)old_message); | | | |
| 41 | } | | | |
| 42 | else { | | | |
| 43 | no_memory_already_linked = FALSE; | | | |
| 44 | } | | | |
| 45 | return(SUCCESS); | | | |
| 46 | } | | | |
| 47 | u_short | | | |
| 48 | is_remove_message(type, str) | | | |
| 49 | u_short type; | | | |
| 50 | char *str; | | | |
| 51 | { | | | |
| 52 | register MESSAGE *message; | | | |
| 53 | register MESSAGE *old_message; | | | |
| 54 | if (start_of_message_list == (MESSAGE *) NULL) { | | | |
| 55 | return(FAILURE); | | | |
| 56 | } | | | |
| 57 | message = start_of_message_list; | | | |
| 58 | old_message = start_of_message_list; | | | |
| 59 | while(message != (MESSAGE *)NULL) { | | | |
| 60 | if ((type == message->message_severity) && | | | |
| 61 | (strcmp(message->message_text, str) == 0)) | | | |
| 62 | { | | | |
| 63 | /* Verify the message we are trying to get rid of */ | | | |
| 64 | old_message->next_message = message->next_message; | | | |
| 65 | if (old_message == message) | | | |
| 66 | start_of_message_list = old_message->next_message; | | | |
| 67 | if (message != test_of_memory) { | | | |
| 68 | (void) free(message->message_text); | | | |
| 69 | (void) free((char *)message); | | | |
| 70 | } | | | |
| 71 | if (--error_count == 0) | | | |
| 72 | error_message_present = FALSE; | | | |
| 73 | return(SUCCESS); | | | |
| 74 | } | | | |
| 75 | old_message = message; | | | |
| 76 | message = message->next_message; | | | |
| 77 | } | | | |
| 78 | return(FAILURE); | | | |
| 79 | } | | | |
| 80 | void | | | |
| 81 | is_message_types(errors, warnings) | | | |
| 82 | u_short *errors; | | | |
| 83 | u_short *warnings; | | | |
| 84 | { | | | |
| 85 | *errors = error_count; | | | |
| 86 | *warnings = warning_count; | | | |
| 87 | } | | | |
| 88 | static void | | | |
| 89 | get_system_string(error_number, str) | | | |
| 90 | int | | | |
| 91 | error_number; | | | |
| 92 | char | | | |
| 93 | *str; | | | |
| 94 | { | | | |
| 95 | iferror VMS | | | |
| 96 | extern int sys_err; | | | |
| 97 | extern char *sys_errlist[]; | | | |
| 98 | if (error_number < 0) | | | |
| 99 | (void) printf(str, ""); | | | |
| 100 | else if (error_number < sys_err) | | | |
| 101 | (void) printf(str, "%s (%d)", sys_errlist[error_number], error_number); | | | |
| 102 | else (void) printf(str, "%s (%d)", error_number); | | | |
| 103 | } | | | |
| 104 | /* ~ VMS ~ */ | | | |
| 105 | { | | | |
| 106 | unsigned short message_length; | | | |
| 107 | unsigned long status; | | | |
| 108 | unsigned long flags = 0x0F; | | | |
| 109 | struct decbdescriptor message_descriptor; | | | |
| 110 | struct { | | | |
| 111 | unsigned char reserved1; | | | |
| 112 | unsigned char free_count; | | | |
| 113 | unsigned char user_value; | | | |
| 114 | unsigned char reserved2; | | | |
| 115 | } status_struct; | | | |
| 116 | message_descriptor.decb.class = DECB_CLASS_S; | | | |
| 117 | message_descriptor.decb.dtype = DECB_DTYPE_T; | | | |
| 118 | message_descriptor.decb.length = 256; /* MAX VMS message */ | | | |
| 119 | message_descriptor.decb.pointer = str; | | | |
| 120 | status = SYSGETMSG(error_number, &message_length, | | | |
| 121 | &message_descriptor, flags, &status_struct); | | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM | DATE | PAGE # |
|--|---|----------------|-----------------|--------|
| | | misc/message.c | 5/23/89 | |
| | | | TIME 1:20:42 pm | 3/3 |
| SOURCE TEXT | | | | |
| LINE # | | | | |
| 241 | if (! (status & 1)) | | | |
| 242 | LIBSIGNAL (status); | | | |
| 243 | | | | |
| 244 | a[message_length] = '\0'; /* null terminate string */ | | | |
| 245 | | | | |
| 246 | } | | | |
| 247 | #endif /* VMS */ | | | |
| 248 | | | | |
| 249 | | | | |
| 250 | | | | |
| 251 | static void | | | |
| 252 | queue_message(severity, str) | | | |
| 253 | u_short severity; | | | |
| 254 | char *str; | | | |
| 255 | { | | | |
| 256 | register int length; | | | |
| 257 | register MESSAGE *message; | | | |
| 258 | | | | |
| 259 | message = (MESSAGE *) malloc((unsigned) sizeof(MESSAGE)); | | | |
| 260 | if (message == (MESSAGE *) NULL) { | | | |
| 261 | handle_out_of_memory(); | | | |
| 262 | return; | | | |
| 263 | } | | | |
| 264 | message->message_severity = severity; | | | |
| 265 | | | | |
| 266 | if ((length - strlen(str)) >= 256) { | | | |
| 267 | length = 256; | | | |
| 268 | str[length] = '\0'; | | | |
| 269 | } | | | |
| 270 | message->message_text = malloc((unsigned)(length+1)); | | | |
| 271 | if (message->message_text == (char *) NULL) { | | | |
| 272 | free((char *) message); | | | |
| 273 | handle_out_of_memory(); | | | |
| 274 | return; | | | |
| 275 | } | | | |
| 276 | (void) strcpy(message->message_text, str); | | | |
| 277 | | | | |
| 278 | link_message(message); | | | |
| 279 | } | | | |
| 280 | | | | |
| 281 | | | | |
| 282 | | | | |
| 283 | | | | |
| 284 | | | | |
| 285 | | | | |
| 286 | static void | | | |
| 287 | handle_out_of_memory() | | | |
| 288 | { | | | |
| 289 | if (no_memory_already_linked == FALSE) { | | | |
| 290 | no_memory_already_linked = TRUE; | | | |
| 291 | out_of_memory.message_severity = ERROR_MSG; | | | |
| 292 | out_of_memory.message_text = no_memory_message; | | | |
| 293 | | | | |
| 294 | link_message(&out_of_memory); | | | |
| 295 | } | | | |
| 296 | | | | |
| 297 | | | | |
| 298 | | | | |
| 299 | static void | | | |
| 300 | link_message(message) | | | |
| 301 | MESSAGE *message; | | | |
| 302 | { | | | |
| 303 | if (start_of_message_list == (MESSAGE *) NULL) { | | | |
| 304 | start_of_message_list = end_of_message_list = message; | | | |
| 305 | } | | | |
| 306 | else { | | | |
| 307 | end_of_message_list->next_message = message; | | | |
| 308 | end_of_message_list = message; | | | |
| 309 | } | | | |
| 310 | | | | |
| 311 | end_of_message_list->next_message = (MESSAGE *) NULL; | | | |
| 312 | | | | |
| 313 | } | | | |

| | | | | | |
|--|--|----------------------------------|--|---------------------|---------------|
| Copyright © 1989 Logic Modeling Systems | | SOURCE PROGRAM misc/resolve.c | | DATE 5/23/89 | PAGE # 1/4 |
| | | | | TIME 11:20:42 pm | |

| LINE # | SOURCE TEXT |
|--------|--|
| 1 | /* SCCS ID: resolve.c rev 3.4 5/9/89 at 15:44:24 */ |
| 2 | #ifndef VMS |
| 3 | #include <sys/types.h> |
| 4 | #include <sys/stat.h> |
| 5 | #ifdef SYS5 |
| 6 | #include <direct.h> |
| 7 | #else |
| 8 | #include <sys/dir.h> |
| 9 | #endif |
| 10 | #include <stdio.h> |
| 11 | #include <ctype.h> |
| 12 | #include "lm_sfi.h" |
| 13 | #include "common.h" |
| 14 | #include "message.h" |
| 15 | |
| 16 | #define max_str 256 |
| 17 | |
| 18 | long check_all_sub_dirs(); |
| 19 | char *getenv(); |
| 20 | |
| 21 | lm_check_path_variable (lm_path) |
| 22 | char *lm_path; |
| 23 | { |
| 24 | char *p, *pos, dir[max_str], pathname[1024]; |
| 25 | char *end_of_string; |
| 26 | long accessible; |
| 27 | char delimiter[max_str]; |
| 28 | long status; |
| 29 | struct stat dir_stat; |
| 30 | |
| 31 | /* get the path to search */ |
| 32 | p = (char *) getenv (lm_path); |
| 33 | if (NULL == p) { |
| 34 | lm_queue_message(ERROR_MSG, "environment variable \"%s\" is not defined ", |
| 35 | lm_path); |
| 36 | return (LM_ERROR); |
| 37 | } |
| 38 | |
| 39 | if ((strlen (p) > sizeof (pathname))) { |
| 40 | lm_queue_message(ERROR_MSG, "environment variable \"%s\" is too long ", lm_path); |
| 41 | return (LM_ERROR); |
| 42 | } |
| 43 | |
| 44 | (void) strcpy (pathname, p); |
| 45 | p = pathname; |
| 46 | end_of_string = pathname + strlen(pathname); |
| 47 | |
| 48 | /* find out if it's a colon or space separated list */ |
| 49 | pos = (char *) strchr (pathname, ":"); |
| 50 | if (NULL == pos) pos = pathname + strlen(pathname); |
| 51 | |
| 52 | (void) strcpy (delimiter, pos, 1); |
| 53 | delimiter[1] = NULL; |
| 54 | |
| 55 | /* look for the file in each element of the path */ |
| 56 | status = LM_SUCCESS; |
| 57 | |
| 58 | do { |
| 59 | pos = (char *) strchr (p, delimiter); |
| 60 | if (NULL == pos) pos = pathname + strlen(pathname); |
| 61 | |
| 62 | (void) strcpy (dir, p, pos-p); |
| 63 | dir[pos-p] = NULL; |
| 64 | p = pos + 1; |
| 65 | |
| 66 | accessible = lm_access(dir, 0); |
| 67 | if (accessible != 0) { |
| 68 | lm_queue_message(WARNING_MSG, "directory \"%s\" is \"%s\" does not exist or cannot be accessed ", dir, lm_path); |
| 69 | status = LM_WARNING; |
| 70 | } else { |
| 71 | stat (dir, &dir_stat); |
| 72 | if (!((dir_stat.st_mode & S_IFDIR))) { |
| 73 | lm_queue_message(WARNING_MSG, "\"%s\" is \"%s\" is not a directory ", dir, lm_path); |
| 74 | status = LM_WARNING; |
| 75 | } |
| 76 | } |
| 77 | } while (pos != end_of_string); |
| 78 | |
| 79 | return (status); |
| 80 | |
| 81 | static int |
| 82 | lm_access (filename, read_flag) |
| 83 | char *filename; |
| 84 | int read_flag; |
| 85 | { |
| 86 | |
| 87 | #ifdef SYS_7 |
| 88 | #define shell_var "SHELL" |
| 89 | #define cegis_shell "/csm/sh" |
| 90 | |
| 91 | /* begin filename resolution. The file is assumed to be either |
| 92 | /* (1) exactly as the user entered it (2) exactly as the user |
| 93 | /* entered it, but with :x instead of /; (3) entirely uppercase |
| 94 | /* on the disk (as distributed by MUI) or (4) entirely lowercase |
| 95 | /* on the disk (default for files created under cegis 3.7). |
| 96 | /* |
| 97 | /* The user may completely ignore case sensitivity problems with |
| 98 | /* the following exceptions: |
| 99 | /* |
| 100 | /* 1) The file is mixed case on the disk |
| 101 | /* 2) The file is on a disk NFSed to a true Unix machine |
| 102 | /* |
| 103 | /* In these cases, the filename must be entered as it resides on |
| 104 | /* the disk. |
| 105 | /* |
| 106 | if (getenv(shell_var) && (0 == strcmp(cegis_shell, getenv(shell_var)))) { |

| Copyright 1989 Logic Modeling Systems | SOURCE PROGRAM misc/resolve.c | DATE 5/23/89 TIME 1:20:42 pm | PAGE # 2/5 |
|--|--|---------------------------------------|---------------|
| LINE # | SOURCE TEXT | | |
| 121 | char local_filename[max_str], /* Don't change the filename in here. */ | | |
| 122 | char old_local_filename[max_str]; | | |
| 123 | char *filename_pointer; | | |
| 124 | char *old_filename; | | |
| 125 | /* Check the filename as entered by the user. If we find */ | | |
| 126 | /* it great, if not, keep hunting */ | | |
| 127 | if (0 == access(filename, read_flag)) | | |
| 128 | return(0); | | |
| 129 | /* convert all uppercase to lower, 'x' to 'X' for all */ | | |
| 130 | /* characters including the path. */ | | |
| 131 | strcpy(local_filename, filename); | | |
| 132 | filename_pointer = local_filename; | | |
| 133 | old_filename = filename; | | |
| 134 | while (*old_filename) { | | |
| 135 | if (isupper(*old_filename)) { | | |
| 136 | *filename_pointer++ = 'x'; | | |
| 137 | *filename_pointer++ = tolower(*old_filename++); | | |
| 138 | } else | | |
| 139 | *filename_pointer++ = *old_filename++; | | |
| 140 | } | | |
| 141 | *filename_pointer = NULL; | | |
| 142 | if (0 == access(local_filename, read_flag)) { | | |
| 143 | strcpy(filename, local_filename); | | |
| 144 | return(0); | | |
| 145 | } | | |
| 146 | /* remove all 'x's (except 'X's) in the filename, NOT */ | | |
| 147 | /* the path, and try again. */ | | |
| 148 | filename_pointer = local_filename; | | |
| 149 | if (strchr(filename_pointer, 'x')) | | |
| 150 | filename_pointer = strchr(filename_pointer, '/'); | | |
| 151 | while (*filename_pointer) { | | |
| 152 | if ((*filename_pointer == 'x') && (*(filename_pointer+1) != '/')) | | |
| 153 | strcpy(filename_pointer, filename_pointer+1); | | |
| 154 | filename_pointer++; | | |
| 155 | } | | |
| 156 | *filename_pointer = NULL; | | |
| 157 | if (0 == access(local_filename, read_flag)) { | | |
| 158 | strcpy(filename, local_filename); | | |
| 159 | return(0); | | |
| 160 | } | | |
| 161 | /* add 'x's to all characters (assume all CAPS) in the */ | | |
| 162 | /* filename NOT the path, and try yet again. */ | | |
| 163 | strcpy(old_local_filename, local_filename); | | |
| 164 | filename_pointer = local_filename; | | |
| 165 | if (strchr(filename_pointer, 'x')) | | |
| 166 | filename_pointer = strchr(filename_pointer, '/'); | | |
| 167 | old_filename = old_local_filename; | | |
| 168 | if (strchr(old_filename, 'x')) | | |
| 169 | old_filename = strchr(old_filename, '/'); | | |
| 170 | while (*old_filename) { | | |
| 171 | if (isalpha(*old_filename)) { | | |
| 172 | *filename_pointer++ = 'x'; | | |
| 173 | *filename_pointer++ = *old_filename++; | | |
| 174 | } | | |
| 175 | *filename_pointer = NULL; | | |
| 176 | if (0 == access(local_filename, read_flag)) { | | |
| 177 | strcpy(filename, local_filename); | | |
| 178 | return(0); | | |
| 179 | } | | |
| 180 | } | | |
| 181 | /* If not running single I/O, just return the access() */ | | |
| 182 | return (access(filename, read_flag)); | | |
| 183 | } | | |
| 184 | } | | |
| 185 | /* First, see if the file is accessible. */ | | |
| 186 | accessible = lm_access(filename, (int) read_flag); | | |
| 187 | (void) strcpy(filename, filename, (int) filename_size); | | |
| 188 | if ((accessible == 0) && (!file_is_a_directory(filename))) return(LM_SUCCESS); | | |
| 189 | /* If it's an absolute path and we're writing the file. */ | | |
| 190 | /* check to see if the directory can be written to. */ | | |
| 191 | if (('/' == filename[0]) && (read_flag == 2)) { | | |
| 192 | (* (char *) strchr(filename, '/')) = (char) NULL; | | |
| 193 | accessible = lm_access(filename, (int) read_flag); | | |
| 194 | if ((accessible == 0) && (!file_is_a_directory(filename))) return(LM_SUCCESS); | | |
| 195 | return(LM_ERROR); | | |
| 196 | } | | |
| 197 | /* get the path to search. */ | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM misc/resolve.c | DATE 5/23/89 | PAGE # 3/6 |
|--|--|---|-----------------|---------------|
| LINE # | | SOURCE TEXT | | |
| 241 | | p = (char *) getenv (lm_path); | | |
| 242 | | if (NULL == p) { | | |
| 243 | | lm_queue_message(ERROR_MSG, "environment variable \"%s\" is not defined ", | | |
| 244 | | lm_path); | | |
| 245 | | return(LM_ERROR); | | |
| 246 | | } | | |
| 247 | | if ((strlen(p) > sizeof(pathname))) { | | |
| 248 | | lm_queue_message(ERROR_MSG, "environment variable \"%s\" is too long ", lm_path); | | |
| 249 | | return(LM_ERROR); | | |
| 250 | | } | | |
| 251 | | (void) strcpy (pathname, p); | | |
| 252 | | p = pathname; | | |
| 253 | | end_of_string = pathname + strlen(pathname); | | |
| 254 | | /* find out if it's a colon or space separated list */ | | |
| 255 | | pos = (char *) strchr (pathname, ":"); | | |
| 256 | | if (NULL == pos) pos = pathname + strlen(pathname); | | |
| 257 | | (void) strcpy (delimiter, pos, 1); | | |
| 258 | | delimiter[1] = NULL; | | |
| 259 | | /* look for the file in each element of the path */ | | |
| 260 | | do { | | |
| 261 | | pos = (char *) strchr (p, delimiter); | | |
| 262 | | if (NULL == pos) pos = pathname + strlen(pathname); | | |
| 263 | | (void) strcpy (p, pos-p); | | |
| 264 | | dir[pos-p] = NULL; | | |
| 265 | | p = pos + 1; | | |
| 266 | | (void) strcpy (dir, "/"); | | |
| 267 | | (void) strcpy (dir, filename); | | |
| 268 | | accessible = lm_access(dir, (int) read_flag); | | |
| 269 | | if ((accessible == 0) && (!file_is_a_directory(dir))) { | | |
| 270 | | (void) strcpy (return_filename, dir, (int) filename_size); | | |
| 271 | | return (LM_SUCCESS); | | |
| 272 | | } | | |
| 273 | | } while (pos != end_of_string); | | |
| 274 | | /* if we get here, the file does not exist */ | | |
| 275 | | /* which is OK if we're going to create it */ | | |
| 276 | | if (2 == read_flag) { | | |
| 277 | | /* return the first directory that we can write to */ | | |
| 278 | | p = (char *) getenv (lm_path); | | |
| 279 | | (void) strcpy (pathname, p); | | |
| 280 | | p = pathname; | | |
| 281 | | pos = (char *) strchr (pathname, ":"); | | |
| 282 | | if (NULL == pos) pos = pathname + strlen(pathname); | | |
| 283 | | (void) strcpy (delimiter, pos, 1); | | |
| 284 | | delimiter[1] = NULL; | | |
| 285 | | do { | | |
| 286 | | pos = (char *) strchr (p, delimiter); | | |
| 287 | | if (NULL == pos) pos = pathname + strlen(pathname); | | |
| 288 | | (void) strcpy (dir, p, pos-p); | | |
| 289 | | dir[pos-p] = NULL; | | |
| 290 | | p = pos + 1; | | |
| 291 | | accessible = lm_access(dir, (int) read_flag); | | |
| 292 | | if (accessible == 0) { | | |
| 293 | | (void) strcpy (dir, "/"); | | |
| 294 | | (void) strcpy (dir, filename); | | |
| 295 | | (void) strcpy (return_filename, dir, (int) filename_size); | | |
| 296 | | return (LM_SUCCESS); | | |
| 297 | | } | | |
| 298 | | } while (pos != end_of_string); | | |
| 299 | | /* if we get here, we couldn't find a directory to write to */ | | |
| 300 | | lm_queue_message(ERROR_MSG, | | |
| 301 | | "write permission denied on file \"%s\" in \"%s\" ", | | |
| 302 | | filename, lm_path); | | |
| 303 | | return(LM_ERROR); | | |
| 304 | | } | | |
| 305 | | lm_queue_message(ERROR_MSG, | | |
| 306 | | "unable to find file \"%s\" in \"%s\" ", filename, lm_path); | | |
| 307 | | return (LM_ERROR); | | |
| 308 | | } | | |
| 309 | | lm_resolve_library_file(lm_path, target_filename, return_filename, filename_size) | | |
| 310 | | char *lm_path, *target_filename, *return_filename, | | |
| 311 | | long filename_size; | | |
| 312 | | { | | |
| 313 | | char *p, *pos, dir[max_str], pathname[1024]; | | |
| 314 | | char file_name[max_str]; | | |
| 315 | | char end_of_string; | | |
| 316 | | long accessible, max_files; | | |
| 317 | | char delimiter[max_str]; | | |
| 318 | | /* first, check to see if the file is in the default directory */ | | |
| 319 | | accessible = lm_access(target_filename, 0); | | |
| 320 | | (void) strcpy (return_filename, target_filename, (int) filename_size); | | |
| 321 | | if ((accessible == 0) && (!file_is_a_directory(target_filename))) return(LM_SUCCESS); | | |
| 322 | | /* Now, let's look down the LM_DIR path */ | | |
| 323 | | p = (char *) getenv (lm_path); | | |
| 324 | | if (NULL == p) { | | |
| 325 | | lm_queue_message(ERROR_MSG, "environment variable \"%s\" is not defined ", | | |
| 326 | | lm_path); | | |
| 327 | | return(LM_ERROR); | | |

| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM misc/resolve.c | # | DATE 5/23/89 | PAGE # 4/7 |
|--|--|--|---|-----------------|---------------|
| LINE # | | SOURCE TEXT | | | |
| 361 | | } | | | |
| 362 | | if ((strlen(p) > sizeof(pathname))) { | | | |
| 363 | | lm_queue_message(ERROR_MSG, "environment variable \"%s\" is too long", lm_path); | | | |
| 364 | | return(LM_ERROR); | | | |
| 365 | | } | | | |
| 366 | | (void) strcpy(pathname, p); | | | |
| 367 | | p = pathname; | | | |
| 368 | | pos = (char *) strchr(pathname, "."); | | | |
| 369 | | if (NULL == pos) pos = pathname + strlen(pathname); | | | |
| 370 | | (void) strcpy(delimiter, pos, 1); | | | |
| 371 | | delimiter[1] = NULL; | | | |
| 372 | | /* look in each of the elements of the path */ | | | |
| 373 | | end_of_string = pathname + strlen(pathname); | | | |
| 374 | | do { | | | |
| 375 | | pos = (char *) strchr(p, delimiter); | | | |
| 376 | | if (NULL == pos) pos = pathname + strlen(pathname); | | | |
| 377 | | (void) strcpy(dir, p, pos-p); | | | |
| 378 | | dir[pos-p] = NULL; | | | |
| 379 | | p = pos + 1; | | | |
| 380 | | (void) strcpy(file_name, dir); | | | |
| 381 | | (void) strcat(file_name, "/"); | | | |
| 382 | | (void) strcat(file_name, target_filename); | | | |
| 383 | | accessible = lm_access(file_name, 0); | | | |
| 384 | | if ((accessible == 0) && (!file_is_a_directory(file_name))) { | | | |
| 385 | | (void) strcpy(return_filename, file_name, (int) filename_size); | | | |
| 386 | | /* we have found the file we're looking for in this */ | | | |
| 387 | | /* directory. Now, check for the same file in a subdi- */ | | | |
| 388 | | /* if we find it, it's an error. */ | | | |
| 389 | | if (0 != check_all_sub_dirs(dir, target_filename, return_filename, filename_size)) { | | | |
| 390 | | lm_queue_message(ERROR_MSG, "%s exists in more than one subdirectory of %s", | | | |
| 391 | | target_filename, dir); | | | |
| 392 | | return(LM_ERROR); | | | |
| 393 | | } | | | |
| 394 | | return(LM_SUCCESS); | | | |
| 395 | | } | | | |
| 396 | | /* We couldn't find the file in the directory, now check all */ | | | |
| 397 | | /* subdirectories. */ | | | |
| 398 | | if (1 == (sum_files = check_all_sub_dirs(dir, target_filename, return_filename, filename_size))) | | | |
| 399 | | return(LM_SUCCESS); | | | |
| 400 | | if (sum_files > 1) { | | | |
| 401 | | if (0 != check_all_sub_dirs(dir, target_filename, return_filename, filename_size)) { | | | |
| 402 | | lm_queue_message(ERROR_MSG, "%s exists in more than one subdirectory of %s", | | | |
| 403 | | target_filename, dir); | | | |
| 404 | | return(LM_ERROR); | | | |
| 405 | | } | | | |
| 406 | | } | | | |
| 407 | | } while (pos != end_of_string); | | | |
| 408 | | lm_queue_message(ERROR_MSG, "unable to find file \"%s\" in \"%s\" ", target_filename, lm_path); | | | |
| 409 | | return(LM_ERROR); | | | |
| 410 | | } | | | |
| 411 | | static long | | | |
| 412 | | check_all_sub_dirs(dir, filename, return_file, ret_size) | | | |
| 413 | | { | | | |
| 414 | | char *dir; | | | |
| 415 | | char *filename; | | | |
| 416 | | char *return_file; | | | |
| 417 | | long ret_size; | | | |
| 418 | | long sum_files = 0; | | | |
| 419 | | char file[max_str]; | | | |
| 420 | | DIR *dir_pointer; | | | |
| 421 | | char *found_dir_name; | | | |
| 422 | | #ifdef SYS5 | | | |
| 423 | | struct direct *dir_entry; | | | |
| 424 | | #else | | | |
| 425 | | struct direct *dir_entry; | | | |
| 426 | | #endif | | | |
| 427 | | dir_pointer = opendir(dir); | | | |
| 428 | | if (NULL == dir_pointer) { | | | |
| 429 | | lm_queue_message(WARNING_MSG, "Unable to open directory for reading: %s", dir); | | | |
| 430 | | return(-1); | | | |
| 431 | | } | | | |
| 432 | | while (NULL != (dir_entry = readdir(dir_pointer))) { | | | |
| 433 | | #ifdef SYS7 | | | |
| 434 | | found_dir_name = dir_entry->d_name + kldge_spollo_sys7_ar9_7(); | | | |
| 435 | | #else | | | |
| 436 | | /* all SYS7 systems */ | | | |
| 437 | | found_dir_name = dir_entry->d_name; | | | |
| 438 | | #endif | | | |
| 439 | | /* SYS7 */ | | | |
| 440 | | if ((strcmp(found_dir_name, ".") != 0) && | | | |
| 441 | | (strcmp(found_dir_name, "..") != 0)) { | | | |
| 442 | | (void) strcpy(file, dir); | | | |
| 443 | | (void) strcat(file, "/"); | | | |
| 444 | | (void) strcat(file, found_dir_name); | | | |
| 445 | | (void) strcat(file, "/"); | | | |
| 446 | | (void) strcat(file, filename); | | | |
| 447 | | if (0 == lm_access(file, 0) && (!file_is_a_directory(file))) { | | | |
| 448 | | sum_files++; | | | |
| 449 | | if (sum_files == 1) { | | | |
| 450 | | (void) strcpy(return_file, file, (int) ret_size); | | | |
| 451 | | if (ret_size < strlen(file)) { | | | |
| 452 | | lm_queue_message(ERROR_MSG, | | | |
| 453 | | "Expanded filename too long for buffer"); | | | |
| 454 | | return(-1); | | | |
| 455 | | } | | | |
| 456 | | } | | | |
| 457 | | } | | | |
| 458 | | } | | | |
| 459 | | } | | | |
| 460 | | } | | | |

| | | | | |
|--|--|----------------------------------|-----------------|---------------|
| Copyright 1989 Logic Modeling Systems | | SOURCE PROGRAM misc/resolve.c | DATE 5/23/89 | PAGE # 5/8 |
| | | TIME 1:20:42 pm | | |

| LINE # | SOURCE TEXT |
|--------|---|
| 481 | |
| 482 | |
| 483 | (void) closedir (dir_pointer); |
| 484 | |
| 485 | return (sum_files); |
| 486 | } |
| 487 | |
| 488 | #ifdef SYS_7 |
| 489 | |
| 490 | /* This routine determines whether we are executing in a SYS or BSD environment |
| 491 | /* by exploiting the fact that on SYS sprintf() returns a reasonable value |
| 492 | /* that equals the number of characters transferred into the buffer. This is |
| 493 | /* opposed to BSD which returns the address of the output buffer. We check to |
| 494 | /* see if sprintf() doesn't return the address of the buffer, thus indicating |
| 495 | /* a SYS runtime environment. |
| 496 | /* |
| 497 | /* This information is used as a component for the difference in the definition |
| 498 | /* of the structure returned by readdir() in the SYS and BSD environments. |
| 499 | /* A lucky(?) turn of events is that in SYS-1, readdir() returns a structure |
| 500 | /* that is equivalent in both environments. |
| 501 | */ |
| 502 | static |
| 503 | kludge_spello_sys_sys_7() |
| 504 | { |
| 505 | char buf[21]; |
| 506 | |
| 507 | if ((int)sprintf(buf, "A") != (int)buf) |
| 508 | return(2); /* SYS */ |
| 509 | else return(0); /* BSD */ |
| 510 | } |
| 511 | #endif /* SYS_7 */ |
| 512 | |
| 513 | |
| 514 | static file_is_a_directory(filename) |
| 515 | char *filename; |
| 516 | |
| 517 | { |
| 518 | struct stat filestat_buffer; |
| 519 | |
| 520 | stat(filename, &filestat_buffer); |
| 521 | if (filestat_buffer.st_mode & S_IFDIR) return (SUCCESS); |
| 522 | |
| 523 | return (FAILURE); |
| 524 | } |
| 525 | |
| 526 | /*else /* VMS */ |
| 527 | |
| 528 | #include "common.h" |
| 529 | #include "message.h" |
| 530 | #include "lm_sfi.h" |
| 531 | #include "sdef.h" |
| 532 | #include "sundef.h" |
| 533 | #include "descrip.h" |
| 534 | |
| 535 | #define max_str 256 |
| 536 | |
| 537 | lm_check_path_variable (lm_path) |
| 538 | char *lm_path; |
| 539 | |
| 540 | { |
| 541 | |
| 542 | if (NULL == (char *) getenv (lm_path)) { |
| 543 | lm_message(ERROR_MSG, "environment variable \"%s\" is not defined ", |
| 544 | lm_path); |
| 545 | return(LM_ERROR); |
| 546 | } |
| 547 | |
| 548 | return(LM_SUCCESS); |
| 549 | } |
| 550 | |
| 551 | |
| 552 | lm_resolve_filename(lm_path, filename, return_filename, filename_size, read_flag) |
| 553 | char *lm_path, *filename, *return_filename; |
| 554 | long filename_size, read_flag; |
| 555 | |
| 556 | { |
| 557 | int context; |
| 558 | int status; |
| 559 | struct descriptor filespec_descriptor; |
| 560 | struct descriptor result_descriptor; |
| 561 | char filespec[max_str]; |
| 562 | struct { |
| 563 | short size; |
| 564 | char result[max_str]; |
| 565 | ret_file; |
| 566 | } |
| 567 | |
| 568 | status = lm_check_path_variable (lm_path); |
| 569 | if (status != LM_SUCCESS) |
| 570 | return(status); |
| 571 | |
| 572 | strcpy (filespec, filename); |
| 573 | |
| 574 | filespec_descriptor.descb_class = DESCX_CLASS_2; |
| 575 | filespec_descriptor.descb_dtype = DESCX_DTYPE_T; |
| 576 | filespec_descriptor.descb_length = strlen(filespec); |
| 577 | filespec_descriptor.descb_pointer = filespec; |
| 578 | |
| 579 | result_descriptor.descb_class = DESCX_CLASS_V2; |
| 580 | result_descriptor.descb_dtype = DESCX_DTYPE_T; |
| 581 | result_descriptor.descb_length = sizeof(ret_file.result); |
| 582 | result_descriptor.descb_pointer = &ret_file; |
| 583 | context = 0; |
| 584 | |
| 585 | status = libsfind_file (&filespec_descriptor, &result_descriptor, &context); |
| 586 | |
| 587 | if ((MSG_NORMAL == status) (MSG_NORMAL == status)) { |
| 588 | ret_file.result[ret_file.size] = '\0'; /* NULL terminate the string */ |
| 589 | strcpy (return_filename, ret_file.result, filename_size); |
| 590 | return(LM_SUCCESS); |
| 591 | } |
| 592 | |
| 593 | strcpy (filespec, lm_path); |
| 594 | strcat (filespec, "/"); |
| 595 | strcat (filespec, filename); |
| 596 | |
| 597 | if (read_flag != 0) { |
| 598 | strcpy (return_filename, filespec, filename_size); |
| 599 | return(LM_SUCCESS); |
| 600 | } |

| COPYRIGHT 1989 Logic Modeling Systems | | SOURCE PROGRAM misc/resolve.c | DATE 5/23/89 | PAGE # 6/9 |
|--|---|----------------------------------|-----------------|---------------|
| LINE # | SOURCE TEXT | | | |
| 601 | filespec_descriptor.dcsb_class = DCSK_CLASS_S, | | | |
| 602 | filespec_descriptor.dcsb_dtyp = DCSK_DTYPE_T, | | | |
| 603 | filespec_descriptor.dcsb_length = strlen(filespec), | | | |
| 604 | filespec_descriptor.dcsb_pointer = filespec, | | | |
| 605 | result_descriptor.dcsb_class = DCSK_CLASS_VS, | | | |
| 606 | result_descriptor.dcsb_dtyp = DCSK_DTYPE_T, | | | |
| 607 | result_descriptor.dcsb_length = sizeof(ret_file.result), | | | |
| 608 | result_descriptor.dcsb_pointer = &ret_file, | | | |
| 609 | count = 0; | | | |
| 610 | status = libfind_file (&filespec_descriptor, &result_descriptor, &count), | | | |
| 611 | if (((XSS_NORMAL == status) (XSS_NORMAL == status))) { | | | |
| 612 | in_queue_message (VMS_ERROR_MSG, status, | | | |
| 613 | "unable to find file \"%s\" in \"%s\" ", filename, | | | |
| 614 | in_path); | | | |
| 615 | return (LM_ERROR); | | | |
| 616 | } | | | |
| 617 | ret_file.result[ret_file.size] = '\0'; /* NULL terminate the string */ | | | |
| 618 | strcpy (return_filename, ret_file.result, filename_size); | | | |
| 619 | return (LM_SUCCESS); | | | |
| 620 | } | | | |
| 621 | } | | | |
| 622 | } | | | |
| 623 | lib_resolve_library_file (in_path, target_filename, return_filename, filename_size) | | | |
| 624 | char *in_path, *target_filename, *return_filename, | | | |
| 625 | long filename_size; | | | |
| 626 | { | | | |
| 627 | int count = 0; | | | |
| 628 | int status; | | | |
| 629 | int count; | | | |
| 630 | char *p; | | | |
| 631 | int i; | | | |
| 632 | struct dcsdescriptor filespec_descriptor; | | | |
| 633 | struct dcsdescriptor result_descriptor; | | | |
| 634 | char filespec[max_str]; | | | |
| 635 | struct { | | | |
| 636 | short size; | | | |
| 637 | char result[max_str]; | | | |
| 638 | ret_file; | | | |
| 639 | } ret_file; | | | |
| 640 | strcpy (filespec, target_filename); | | | |
| 641 | filespec_descriptor.dcsb_class = DCSK_CLASS_S, | | | |
| 642 | filespec_descriptor.dcsb_dtyp = DCSK_DTYPE_T, | | | |
| 643 | filespec_descriptor.dcsb_length = strlen(filespec), | | | |
| 644 | filespec_descriptor.dcsb_pointer = filespec, | | | |
| 645 | result_descriptor.dcsb_class = DCSK_CLASS_VS, | | | |
| 646 | result_descriptor.dcsb_dtyp = DCSK_DTYPE_T, | | | |
| 647 | result_descriptor.dcsb_length = sizeof(ret_file), | | | |
| 648 | result_descriptor.dcsb_pointer = &ret_file, | | | |
| 649 | status = libfind_file (&filespec_descriptor, &result_descriptor, &count), | | | |
| 650 | if (((XSS_NORMAL == status) (XSS_NORMAL == status))) { | | | |
| 651 | ret_file.result[ret_file.size] = '\0'; /* NULL terminate the string */ | | | |
| 652 | strcpy (return_filename, ret_file.result, filename_size); | | | |
| 653 | return (LM_SUCCESS); | | | |
| 654 | } | | | |
| 655 | else { | | | |
| 656 | strcpy (filespec, in_path); | | | |
| 657 | strcat (filespec, "[000000....]"); | | | |
| 658 | strcat (filespec, target_filename); | | | |
| 659 | filespec_descriptor.dcsb_class = DCSK_CLASS_S, | | | |
| 660 | filespec_descriptor.dcsb_dtyp = DCSK_DTYPE_T, | | | |
| 661 | filespec_descriptor.dcsb_length = strlen(filespec), | | | |
| 662 | filespec_descriptor.dcsb_pointer = filespec, | | | |
| 663 | count = 0; | | | |
| 664 | count = 0; | | | |
| 665 | do { | | | |
| 666 | status = libfind_file (&filespec_descriptor, &result_descriptor, &count); | | | |
| 667 | if (((XSS_NORMAL == status) (XSS_NORMAL == status))) { | | | |
| 668 | ret_file.result[ret_file.size] = '\0'; /* NULL terminate the string */ | | | |
| 669 | strcpy (return_filename, ret_file.result, filename_size); | | | |
| 670 | count++; | | | |
| 671 | } while (count != 0); | | | |
| 672 | if (count == 0) { | | | |
| 673 | in_queue_message (VMS_ERROR_MSG, status, | | | |
| 674 | "unable to find file \"%s\" in \"%s\" ", | | | |
| 675 | target_filename, in_path); | | | |
| 676 | return (LM_ERROR); | | | |
| 677 | } | | | |
| 678 | if (count > 1) { | | | |
| 679 | in_queue_message (ERROR_MSG, | | | |
| 680 | "%s exists in more than one subdirectory of %s ", | | | |
| 681 | target_filename, in_path); | | | |
| 682 | return (LM_ERROR); | | | |
| 683 | } | | | |
| 684 | if (count == 1) { | | | |
| 685 | for (i=0; p=return_filename; i<strlen(return_filename); i++, p++) | | | |
| 686 | if (0 == strcmp(p, "[000000....]")) | | | |
| 687 | strcpy (p, p+3); | | | |
| 688 | for (i=0; p=return_filename; i<strlen(return_filename); i++, p++) | | | |
| 689 | if (0 == strcmp(p, "[000000....]")) | | | |
| 690 | strcpy (p, p+3); | | | |
| 691 | return (LM_SUCCESS); | | | |
| 692 | } | | | |
| 693 | } | | | |
| 694 | } | | | |
| 695 | } | | | |
| 696 | } | | | |
| 697 | } | | | |
| 698 | } | | | |
| 699 | } | | | |
| 700 | } | | | |
| 701 | } | | | |
| 702 | } | | | |
| 703 | } | | | |
| 704 | } | | | |
| 705 | } | | | |
| 706 | } | | | |
| 707 | } | | | |
| 708 | } | | | |
| 709 | } | | | |
| 710 | } | | | |
| 711 | } | | | |
| 712 | } | | | |
| 713 | } | | | |
| 714 | } | | | |
| 715 | } | | | |
| 716 | } | | | |
| 717 | } | | | |

We claim:

1. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the

65

hardware modeling system is in a driving low state, the pin driver being individually programmable by software means to drive the pin according to one of a pre-determined soft-drive high I/V characteristic curve and

programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive low I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive low I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the voltage of the pin is below a first reference voltage to indicate that the pin is in the driving low state, or if the voltage of the pin is between the first reference voltage and a second reference voltage to indicate that the pin is in the non-driving state.

8. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state or non-driving state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive high I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive high I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the voltage of the pin is above a first reference voltage to indicate that the pin is in the driving high state, or if the voltage of the pin is between the first reference voltage and a second reference voltage to indicate that the pin is in the non-driving state.

9. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, the pin driver being individually programmable by software means to drive the pin according to one of a predetermined soft-drive high I/V characteristic curve and a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive low I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive low I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than the reference current to indicate that the pin is in a driving low state, or if the current into the pin is less than the reference current to indicate that the pin is in another state.

10. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin according to one of a predetermined soft-drive high I/V characteristic curve and a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

ware means to drive the pin according to one of a predetermined soft-drive high I/V characteristic curve and a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive low I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive low I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is between a first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is other than between the first and second reference currents to indicate that the pin is in another state.

11. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive high I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive high I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is between a first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is other than between the first and second reference currents to indicate that the pin is in another state.

12. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive high I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive high I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is less than a reference current to indicate that the pin is in a driving high state, or if the current into the pin is greater than the reference current to indicate that the pin is in another state.

13. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state,

non-driving state, or driving high state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive low I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive low I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a first reference current to indicate that the pin is in the driving low state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is less than the second reference current to indicate that the pin is in the driving high state.

14. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, non-driving state, or driving high state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive high I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive high I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a first reference current to indicate that the pin is in the driving low state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is less than the second reference current to indicate that the pin is in the driving high state.

15. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state or non-driving state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive low I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive low I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a first reference current to indicate that the pin is in the driv-

ing low state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state.

16. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state or non-driving state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive high I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive high I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is less than a first reference current to indicate that the pin is in the driving high state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state.

17. The method as recited in claim 1, 2, 3, 4, 5, 6, 7, or 8, wherein the method further comprises the step of selecting one of the soft-drive low I/V characteristic curve and the soft-drive high I/V characteristic curve to program the pin driver according to a state of simulated circuitry connected to the pin in a simulated circuit design.

18. The method as recited in claim 9, 10, 11, 12, 13, 14, 15, or 16, wherein the method further comprises the step of selecting one of the soft-drive low I/V characteristic curve and the soft-drive high I/V characteristic curve to program the pin driver according to a state of simulated circuitry connected to the pin in a simulated circuit design.

19. The method as recited in claim 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, or 16, wherein the method further comprises the step of determining simultaneously the states of a plurality of pins of an electronic device or circuitry that are electrically coupled to software programmable pin drivers of the hardware modeling systems.

20. The method as recited in claim 19, wherein the pin drivers that are electrically coupled to the pins are collectively programmable to drive with the predetermined soft-drive high I/V characteristic curve or predetermined soft-drive low I/V characteristic curve.

21. The method as recited in claim 1, 2, 3, 4, 5, 6, 7, or 8, wherein the reference voltages are software programmable by the hardware modeling system.

22. The method as recited in claim 17, wherein the reference voltages are software programmable by the hardware modeling system.

23. The method as recited in claim 19, wherein the reference voltages are software programmable by the hardware modeling system.

24. The method as recited in claim 20, wherein the reference voltages are software programmable by the hardware modeling system.

25. The method as recited in claim 1, 2, 3, 4, 5, 6, 7, or 8, wherein reference voltages are hardware programmable by the hardware modeling system.

26. The method as recited in claim 17, wherein the reference voltages are hardware programmable by the hardware modeling system.

27. The method as recited in claim 19, wherein the reference voltages are hardware programmable by the hardware modeling system.

28. The method as recited in claim 20, wherein the reference voltages are hardware programmable by the hardware modeling system.

29. The method as recited in claim 9, 10, 11, 12, 13, 14, 15, or 16, wherein the reference currents are software programmable by the hardware modeling system.

30. The method as recited in claim 18, wherein the reference currents are software programmable by the hardware modeling system.

31. The method as recited in claim 19, wherein the reference currents are software programmable by the hardware modeling system.

32. The method as recited in claim 20, wherein the reference currents are software programmable by the hardware modeling system.

33. The method as recited in claim 9, 10, 11, 12, 13, 14, 15, or 16, wherein the reference currents are hardware programmable by the hardware modeling system.

34. The method as recited in claim 18, wherein the reference currents are hardware programmable by the hardware modeling system.

35. The method as recited in claim 19, wherein the reference currents are hardware programmable by the hardware modeling system.

36. The method as recited in claim 20, wherein the reference currents are hardware programmable by the hardware modeling system.

37. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the voltage of the pin is below a reference voltage to indicate that the pin is in the driving low state, or if the voltage of the pin is above the reference voltage to indicate that the pin is in another state.

38. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin toward one of a logic low and a logic high, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the voltage of the pin is between a first and a second reference voltage to indicate that the pin is in the non-driving state, or if the voltage of the pin is other than between the first and second reference voltages to indicate that the pin is in another state.

39. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin toward one of a logic low and a logic high, comprising the steps of:

circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin toward one of a logic low and a logic high, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the voltage of the pin is between a first and a second reference voltage to indicate that the pin is in the non-driving state, or if the voltage of the pin is other than between the first and second reference voltages to indicate that the pin is in another state.

40. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state, the pin driver being individually programmable by software means to drive the pin towards a logic low or a logic high, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the voltage of the pin is above a reference voltage to indicate that the pin is in the driving high state, or if the voltage of the pin is below the reference voltage to indicate that the pin is in another state.

41. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, non-driving state, or driving high state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the voltage of the pin is below a first reference voltage to indicate that the pin is in the driving low state, or if the voltage of the pin is between the first and a second reference voltage to indicate that the pin is in the non-driving state, or if the voltage of the pin is above the second reference voltage to indicate that the pin is in the driving high state.

42. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, non-driving state, or driving high state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the voltage of the pin is below a first reference voltage to indicate that the pin is in the driving low state, or if the voltage of the pin is between a first

and a second reference voltage to indicate that the pin is in the non-driving state, or if the voltage of the pin is above the second reference voltage to indicate that the pin is in the driving high state.

43. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state or non-driving state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the voltage of the pin is below a first reference voltage to indicate that the pin is in the driving low state, or if the voltage of the pin is between the first and a second reference voltage to indicate that the pin is in the non-driving state.

44. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state or non-driving state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the voltage of the pin is above a first reference voltage to indicate that the pin is in the driving high state, or if the voltage of the pin is between the first and a second reference voltage to indicate that the pin is in the non-driving state.

45. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in the driving low state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a reference current to indicate that the pin is in the driving low state, or if the current into the pin is less than the reference current to indicate that the pin is in another state.

46. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin toward a logic low or a logic high, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the current into the pin is between a first reference

current and a second reference current to indicate that the pin is in the non-driving state, or if the current at the pin is other than between the first and second reference currents to indicate that the pin is in another state.

47. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin toward a logic low or a logic high, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the current into the pin is between a first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is other than between the first and second reference currents to indicate that the pin is in another state.

48. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state, the pin driver being individually programmable by software means to drive the pin toward a logic low or a logic high, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the current into the pin is less than a reference current to indicate that the pin is in the driving high state, or if the current into the pin is greater than the reference current to indicate that the pin is in another state.

49. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, non-driving state, or driving high state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a first reference current to indicate that the pin is the driving low state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is less than the second reference current to indicate that the pin is in the driving high state.

50. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, non-driving state, or driving high state, the pin driver being individually programmable by software means to

drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a first reference current to indicate that the pin is in the driving low state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is less than the second reference current to indicate that the pin is in the driving high state.

51. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state or non-driving state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a first reference current to indicate that the pin is in the driving low state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state.

52. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state or non-driving state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the current into the pin is less than a first reference current to indicate that the pin is in the driving high state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state.

53. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing the voltage of the pin with a plurality of reference voltage levels that differentiate between at least four states of the pin.

54. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing the current into the pin with a plurality of reference current levels that differentiate between at least four states of the pin.

55. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware

modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time; automatically comparing the voltage at the pin with a plurality of reference voltage levels that differentiate between at least a driving low state, non-driving low state, non-driving high state, and driving high state.

56. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing the current into the pin with a plurality of reference current levels that differentiate between at least a driving low state, non-driving low state, non-driving high state, and driving high state.

57. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing the voltage at the pin with at least three reference voltages.

58. The method as recited in claim 57, wherein the method further includes the step of programming the reference voltages with the hardware modeling system.

59. The method as recited in claim 58, wherein the method includes programming the reference voltages by software means.

60. The method as recited in claim 58, wherein the method includes programming the reference voltages by hardware means.

61. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing the current into the pin with at least three reference currents.

62. The method as recited in claim 61, wherein the method further includes the step of programming the reference currents with the hardware modeling systems.

63. The method as recited in claim 62, wherein the method includes programming the reference currents by software means.

64. The method as recited in claim 62, wherein the method includes programming the reference currents by hardware means.

65. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing voltage at the pin in at least three comparators with a different reference voltage being provided to each comparator.

66. The method as recited in claim 65, wherein the reference voltages are software programmable by the hardware modeling system.

67. The method as recited in claim 65, wherein the reference voltages are hardware programmable by the hardware modeling system.

68. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing the current into the pin

in at least three comparators with a different reference current being provided to each comparator.

69. The method as recited in claim 68, wherein the reference currents are software programmable by the hardware modeling system.

70. The method as recited in claim 68, wherein the reference currents are hardware programmable by the hardware modeling system.

71. A method for use in a hardware modeling system for determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a software programmable, current limited pin driver of the hardware modeling system, with the pin driver being programmable to drive with any one of at least eight different current limits, and at the same time, automatically comparing the voltage of the pin with at least one reference voltage.

72. The method as recited in claim 71 wherein the electronic devices or circuitry includes a plurality of pins electrically coupled to pin drivers of the hardware modeling system, and the current limit of the pin driver associated with each pin is independently software programmable by the hardware modeling system.

73. A method for use in a hardware modeling system for determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a software programmable, current limited pin driver of the hardware modeling system, with the pin driver being programmable to drive with any one of at least eight different current limits and at the same time, automatically comparing the current into the pin with at least one reference current.

74. The method as recited in claim 73, wherein the electronic devices or circuitry includes a plurality of pins electrically coupled to pin drivers of the hardware modeling system, and the current limit of the pin driver associated with each pin is independently software programmable by the hardware modeling system.

75. A method for use in a hardware modeling system for determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a software programmable, current limited pin driver of the hardware modeling system, the pin driver being both a current sink and source and driving toward a predetermined software programmable voltage level, and at the same time, automatically comparing the voltage of the pin with at least two reference voltages.

76. A method for use in a hardware modeling system for determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a software programmable, current limited pin driver of the hardware modeling system, the pin driver being both a current sink and source and driving toward a predetermined software programmable voltage level and at the same time, automatically comparing the current into the pin with at least two reference currents.

77. A method for use in a hardware modeling system for determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a software programmable pin driver of the hardware modeling system, the pin driver being programmable to drive with any one of at least eight different I/V characteristic curves, and at the same time, automatically

comparing the voltage of the pin with at least one reference voltage.

78. A method for use in a hardware modeling system for determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a software programmable pin driver of the hardware modeling system, the pin driver being programmable to drive with any one of at least eight different I/V characteristic curves, and at the same time, automatically comparing the current into the pin with at least one reference current.

79. The method as recited in claims 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, wherein the method includes determining the state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system in response to an access to the hardware modeling system being assessed by a simulator via network means.

80. The method as recited in claims 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, wherein the method includes determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system with the hardware modeling system being time shared among a plurality of simulators.

81. The method as recited in claims 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, wherein the method includes determining the state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, the hardware modeling system having a plurality of electronic devices or circuitry connected thereto with each electronic device or circuitry having at least one pin that is electrically coupled to the hardware modeling system.

82. The method as recited in claims 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, wherein the pin is capable of functioning as both an input and an output.

83. The method as recited in claims 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, wherein the method includes determining the states of a plurality of pins of the electronic device or circuitry simultaneously.

84. The method as recited in claim 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, or 78, wherein the method further includes as a first step resetting and restoring the electronic device or circuitry to a specific internal state by presentation of a history sequence of stimulation patterns to the electronic device or circuitry.

85. A multi-channel pin driver integrated circuit of a hardware modeling system for simultaneously stimulating and sensing at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including at least one current limited pin driver of the hardware modeling system that is electrically coupled to a pin of the electronic device or circuitry, the current limit of the pin driver of the hardware modeling system being software programmable by the hardware modeling system.

86. A multi-channel pin driver integrated circuit of a hardware modeling system for simultaneously stimulating and sensing at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including at least one pin driver of the hardware modeling system that is electrically coupled to a pin of the electronic device or circuitry, the pin driver of the hardware mod-

eling system providing a programmable drive voltage for driving the pin connected thereto, the drive voltage being programmable to at least three different voltages.

87. The multi-channel pin driver integrated circuit as recited in claim 86, wherein the drive voltage is software programmable by the hardware modeling system.

88. A multi-channel pin driver integrated circuit of a hardware modeling system for simultaneously stimulating and sensing at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including at least one channel having at least three voltage comparators for use in determining a state of at least one pin.

89. A multi-channel pin driver integrated circuit of a hardware modeling system for simultaneously stimulating and sensing at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including at least one channel having at least one voltage comparator for use in determining the state of at least one pin, the voltage comparator having input thereto a programmable reference voltage.

90. The multi-channel pin driver integrated circuit as recited in claim 89, wherein the reference voltage is software programmable by the hardware modeling system.

91. A method for use in a hardware modeling system for executing hardware modeling system diagnostics using known-good diagnostic circuitry, comprising the steps of:

- (a) electrically coupling the diagnostic circuitry to the pin electronics circuitry of the hardware modeling system;
- (b) presenting stimulus patterns to the diagnostic circuitry using the pin electronics circuitry;
- (c) measuring a response of the diagnostic circuitry to the stimulus patterns applied in step (b); and
- (d) automatically comparing the response measured in step (c) with a known-good response to determine if the hardware modeling system is functioning properly.

92. A method for use in a hardware modeling system for determining a state of at least one pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system, and at the same time, automatically determining whether the pin is in a driving low state or a non-driving low state.

93. A method for use in a hardware modeling system for determining a state of at least one pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system, and at the same time, automatically determining whether the pin is in a driving high state or a non-driving high state.

94. A method for use in a hardware modeling system for restoring an electronic device or circuitry to a specific internal state, the electronic device or circuitry having at least one I/O pin electrically coupled to the hardware modeling system, the method comprising presenting a history sequence of stimulation patterns to the I/O pin of the electronic device or circuitry with a current limited pin driver of the hardware modeling system coupled to the I/O pin.

95. The method as recited in claim 94, wherein the current limit for the pin driver is less than 50 mA.

96. A method for use in a hardware modeling system for restoring an electronic device or circuitry to a specific internal state, the electronic device or circuitry having at least one I/O pin electrically coupled to the hardware modeling system, the method comprising presenting a history sequence of stimulation patterns to the I/O pin of the electronic device or circuitry with a pulsed driver of the hardware modeling system coupled to the I/O pin.

97. A method for use in a hardware modeling system for determining an output delay of at least a first pin of an electronic device or circuitry according to a present internal state of the electronic device or circuitry in response to a stimulus applied to at least a second pin of the electronic device or circuitry, the pins being electrically coupled to the hardware modeling system, comprising the steps of:

- (a) resetting the electronic device or circuitry coupled to the hardware modeling system to a known internal state;
- (b) restoring the electronic device or circuitry coupled to the hardware modeling system to the present internal state;
- (c) stimulating the electronic device or circuitry coupled to the hardware modeling system by applying stimulus through at least the second pin;
- (d) sampling the state of the first pin according to a software programmable delay after the stimulus was applied;
- (e) changing the software programmable delay in a predetermined manner based on the state sampled during step (d); and
- (f) repeating steps (a)–(e) until the output delay time is determined.

98. A multi-channel pin driver integrated circuit of a hardware modeling system for stimulating at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including circuit means to stage pin stimulation patterns for simultaneous presentation of the pin stimulation patterns to the pins connected to the integrated circuit and for simultaneous presentation of the pin stimulation patterns with pin stimulation patterns of other multi-channel pin driver integrated circuits.

99. A multi-channel pin driver integrated circuit of a hardware modeling system for stimulating at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including one shared data path for both programming internal control registers of the integrated circuit and for providing pin stimulation pattern data to the integrated circuit from circuitry of the hardware modeling system.

100. A multi-channel pin driver integrated circuit of a hardware modeling system for stimulating at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including one shared data path for both reading internal control registers of the integrated circuit and for providing pin stimulation pattern data to the integrated circuit from circuitry of the hardware modeling system.

101. A multi-channel pin driver integrated circuit of a hardware modeling system for stimulating at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including circuitry for error checking incoming pins stimulation patterns.

102. The multi-channel pin driver integrated circuit as recited in claim 101, wherein the error checking circuitry performs parity error checks.

103. A multi-channel pin driver integrated circuit of a hardware modeling system for stimulating at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including a strobe input for use in determining the sampling times for measuring output delays.

104. A method for use in a hardware modeling system of restoring an internal state of an electronic device or circuitry having at least one I/O pin, the I/O pin being electrically coupled to a pin driver of the hardware modeling system, comprising the steps of:

- (a) storing a history sequence of stimulation patterns for the I/O pin in memory means with at least one stimulation pattern of the history sequence for the I/O pin being stored in a single bit of the memory means;
- (b) retrieving from the memory means at least one stimulation pattern of the history sequence for the I/O pin;
- (c) presenting the retrieved I/O pin stimulation pattern or patterns to the I/O pin with the pin driver of the hardware modeling system; and
- (d) repeating steps (b) and (c) until the entire history sequence is presented to the I/O pin.

105. A method for use in a hardware modeling system of restoring an internal state of an electronic device or circuitry having at least one I/O pin, the I/O pin being electronically coupled to a pin driver of the hardware modeling system, comprising the steps of:

- (a) storing a history sequence of stimulation patterns for the I/O pin in memory means in a plurality of memory means bits, the number of bits being used being less than twice the number of stimulation patterns for the I/O pin;
- (b) retrieving from the memory means at least one stimulation pattern of the history sequence for the I/O pin;
- (c) presenting the retrieved I/O pin stimulation pattern or patterns to the I/O pin with the pin driver of the hardware modeling system; and
- (d) repeating steps (b) and (c) until the entire history sequence is presented to the pin.

106. A method for use in a hardware modeling system of restoring an internal state of an electronic device or circuitry having pins electrically coupled to pin drivers of the hardware modeling system and at least one I/O pin that is electrically coupled to a pin driver of the hardware modeling system, comprising the steps of:

- (a) presenting a history sequence of stimulation patterns to the pins of the electronic device or circuitry with the pin drivers of the hardware modeling system;
- (b) determining the state of the I/O pin;
- (c) embedding in the history sequence a stimulation pattern which will cause the pin driver of the hardware modeling system connected to the I/O pin to

drive high when said pattern of the history sequence is again presented regardless of the logic state of simulated circuitry connected to the I/O pin in a design under simulation if the state of the I/O pin was determined in step (b) to be a driving high state; and

- (d) embedding in the history sequence a pin stimulation pattern which will cause the pin driver of the hardware modeling system connected to the I/O pin to drive in a low state when said pattern of the history sequence is again presented regardless of the logic state of a simulated circuitry connected to the I/O pin in a design under simulation if the state of the I/O pin was determined in step (b) to be a driving low state.

107. A hardware modeling system for stimulating and sensing a response of electronic devices or circuitry to the stimulus, the electronic device or circuitry being electrically coupled to the hardware modeling system, the improvement being that the presence and type of electronic device or circuitry is determined automatically when the electronic device or circuitry is connected to a powered hardware modeling system.

108. A hardware modeling system for stimulating and sensing a response of electronic devices or circuitry to the stimulus, the improvement being that the hardware modeling system has fixturing means for connecting an electronic device or circuitry to the hardware modeling system that is powered.

109. An apparatus for connecting an electronic device or circuitry to a hardware modeling system, comprising fixturing means that has matched length traces connecting pins of the electronic device or circuitry to pin electronics circuitry of the hardware modeling system.

110. A method for use in a hardware modeling system for generating a portion of an electronic device or circuitry timing specification, the electronic device or circuitry being electrically coupled to the hardware modeling system, the method comprising measuring output delays of pins of the electrical device or circuitry coupled to the hardware modeling system, in response to stimulus, and deriving the timing specification from the output delays.

111. A hardware modeling system for stimulating and sensing a response of electronic devices or circuitry to the stimulus, the hardware modeling system having a plurality of electronic devices or circuitry electrically coupled thereto, the improvement being that a fixturing means of the hardware modeling system provides a plurality of different power supply voltages to accommodate connections of electronic devices or circuitry that operate at different power supply voltages.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO. : 5,353,243
DATED : October 4, 1994
INVENTOR(S) : Andrew J. Read et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 1863, line 48, change "low" to --high--.
Column 1863, line 58, insert --the-- after "that".
Column 1865, line 59, change "that" to --than--.
Column 1865, line 62, insert --that-- after "indicate".
Column 1868, line 67, insert --the-- after "wherein".
Column 1871, line 52, change "circuit" to --current--.
Column 1874, lines 42-43, change "systems" to
--system--.
Column 1875, line 58, insert a comma --,-- immediately
after "level".
Column 1878, line 68, change "pins" to --pin--.

Signed and Sealed this
Twenty-first Day of March, 1995

Attest:



BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks